

國立臺灣大學電機資訊學院資訊工程研究所

碩士論文

Department of Computer Science & Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

探討以太坊代理合約所引發的安全問題

Investigating New Security Issues Introduced by Ethereum  
Proxy Contracts

朱玟嶧

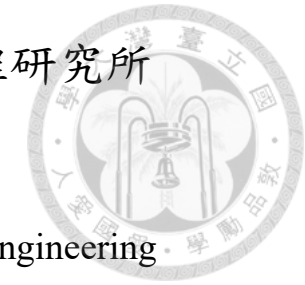
Wen-Yi Chu

指導教授: 蕭旭君 博士

Advisor: Hsu-Chun Hsiao, Ph.D.

中華民國 111 年 9 月

September, 2022





國立臺灣大學碩士學位論文  
口試委員會審定書

探討以太坊代理合約所引發的安全問題

Investigating New Security Issues Introduced by  
Ethereum Proxy Contracts

本論文係朱玟嶧君（學號 R09922070）在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國 111 年 8 月 2 日承下列考試委員審查通過及口試及格，特此證明

口試委員：

蕭旭君

陳昱圻

（指導教授）

廖世偉

郭博鈞

系主任

洪士灝



## 致謝

這篇論文的完成首先要感謝的是蕭旭君老師的指導，如果沒有老師在每週開會時給的種種建議和提點，這篇論文就無法產出。

感謝蕭旭君老師在發想題目時給我方向，並協助我想出實驗方法以及完成實驗。對於我論文初稿的修正也是盡力地幫忙，原本的文句，經過蕭老師的潤飾修改，讓論文的可讀性提高很多，讓讀者在閱讀時更能了解論文的脈絡。蕭老師在口試的準備也給予我很多的建議，讓我的口試內容更加流暢。蕭老師真的十分的盡責，身為學生的我除了專業知識外，更是從老師身上學到了努力負責的工作態度及嚴謹的研究精神。

感謝實驗室同學們一路上的陪伴，我們彼此互相督促，感謝吳家謙和謝立峴兩位同學在每周開會時給予我建議，點出我不曾思考過的問題。感謝李洵、許育銘、吳家謙同學給予我準備口試上的建議。

最後感謝我的家人，一直在我身旁鼓勵我，提供我生活所需的開銷，讓我讀碩士的期間沒有後顧之憂。



## 摘要

由於區塊鏈的不可更改性，智能合約一旦發布後，開發者便無法修正含有漏洞的智能合約，為了能夠讓開發者修正已經發布的智能合約，因此而產生了代理模式的開發方式。代理合約負責儲存一個智能合約的地址，而實際上執行的程式邏輯都在這一個智能合約中，當我們要修正智能合約時，我們只要更改代理合約儲存的地址，即可讓代理合約執行另一個智能合約的程式邏輯，達到修正漏洞的效果。然而，這樣的開發方式也帶來了新的安全問題，分別是函數名稱衝突以及儲存位置衝突。為了探討代理合約在以太坊上面的影響，我們製作了一個分析代理合約的工具 ProxyChecker。我們使用 ProxyChecker 分析六段時間的智能合約，並發現代理合約有越來越多的趨勢，從 2017 年的 1% 到 2022 年的 88%。在這些代理合約中，大部分都有函數名稱衝突的問題，另外有 0 到 3% 有儲存位置衝突的問題。在論文最後，我們提出一些針對代理合約建議的使用方式給開發者和使用者。我們認為使用代理合約最安全的方式是建立自己的代理合約，像是建立自己的代理合約錢包，但是這與代理合約想要升級有問題的合約這個初衷不符。我們認為透過延遲升級的方式可以達到安全性和使用性上的平衡。

**關鍵字：**代理合約、邏輯合約、可升級性、安全、區塊鏈



# Abstract

The immutable feature of blockchains prevents developers from fixing buggy smart contracts. Consequently, the concept of proxy pattern has emerged to support upgradability. By putting the actual program logic into a secondary contract called a logic contract, a proxy contract can be upgraded by switching to a different logic contract. However, the proxy pattern also brings two security issues, function collisions and storage collisions. To examine the effect of proxy contracts on the Ethereum mainnet, we created a tool named ProxyChecker for analyzing proxy contracts. Using ProxyChecker, we analyzed contracts within six block ranges and found that proxy contracts have become more prevalent, from 1% in 2017 to 88% in 2022. Among these proxy contracts we found, the majority have function collisions, and about 0-3% have storage collisions. Lastly, we provide suggestions for developers and users. Although the most secure way to use proxy contracts is for personal use only, such as constructing wallet contracts, this contradicts the original motivation of introducing them, that is, for upgrading and fixing buggy contracts. We

concluded that a delayed upgrade might provide a good balance between security and functionality.



**Keywords:** proxy contract, logic contract, upgradability, security, blockchain



# Contents

	<b>Page</b>
<b>Verification Letter from the Oral Examination Committee</b>	<b>i</b>
<b>致謝</b>	<b>ii</b>
<b>摘要</b>	<b>iii</b>
<b>Abstract</b>	<b>iv</b>
<b>Contents</b>	<b>vi</b>
<b>List of Figures</b>	<b>ix</b>
<b>List of Tables</b>	<b>x</b>
<b>List of Algorithms</b>	<b>xi</b>
<b>Chapter 1 Introduction</b>	<b>1</b>
<b>Chapter 2 Background</b>	<b>6</b>
2.1 EVM . . . . .	6
2.2 EVM Opcode . . . . .	7
2.3 EVM Storage Layout . . . . .	8
2.4 Function Signature . . . . .	8
2.5 Fallback Function . . . . .	9
2.6 Ethereum Improvement Proposals . . . . .	9



<b>Chapter 3</b>	<b>Related Work</b>	<b>12</b>
3.1	Smart Contract Analysis Tools . . . . .	12
3.2	Ethereum Virtual Machine Emulator . . . . .	13
3.2.1	Go-Ethereum . . . . .	13
3.2.2	E-EVM . . . . .	14
3.3	Proxy Contract Related . . . . .	14
<b>Chapter 4</b>	<b>Security Issues</b>	<b>15</b>
4.1	Definition of Proxy Contracts . . . . .	15
4.2	Function Collisions . . . . .	15
4.3	Storage Collisions . . . . .	18
<b>Chapter 5</b>	<b>PROXYCHECKER: Design and Implementation</b>	<b>21</b>
5.1	Overview . . . . .	21
5.2	Delegatecall Detector . . . . .	23
5.3	Dynamic Analyzer . . . . .	24
5.3.1	Create a Calldata . . . . .	25
5.3.2	Dynamically Analyze and Extract the Parameters . . . . .	26
5.3.3	Compare the Calldata . . . . .	28
5.4	Contract Checker . . . . .	29
<b>Chapter 6</b>	<b>Data Collection and Analysis Result</b>	<b>34</b>
<b>Chapter 7</b>	<b>In-depth Analysis</b>	<b>41</b>
7.1	Logic Contracts . . . . .	42
7.2	Proxy Contracts . . . . .	46
7.3	Other Findings . . . . .	48



<b>Chapter 8</b>	<b>DISCUSSION</b>	<b>50</b>
8.1	Mitigation for Security Issues . . . . .	50
8.2	Proxy Contract Usage . . . . .	52
8.3	Limitations . . . . .	54
<b>Chapter 9</b>	<b>CONCLUSION</b>	<b>55</b>
<b>References</b>		<b>57</b>





# List of Figures

2.1	EVM components . . . . .	6
5.1	ProxyChecker overview . . . . .	22
5.2	Delegatecall Detector . . . . .	24
5.3	Dynamic Analyzer . . . . .	25
5.4	Contract Checker . . . . .	29
6.1	The query we used in BigQuery . . . . .	35
6.2	Result of Dynamic Analyzer . . . . .	37



# List of Tables

5.1	Values for block opcodes . . . . .	28
6.1	Dataset information . . . . .	35
6.2	Number and percentage of contracts of each type in different datasets. . .	36
6.3	The number of contracts classified by where logic contract address is stored	38
6.4	Number of contracts is verified among EIP-1822, EIP-1967 and Others .	39
6.5	Number of contracts of collision issues among verified contracts . . . . .	39
7.1	The top 5 most pointed logic contracts (part 1). . . . .	41
7.2	The top 5 most pointed logic contracts (part 2). . . . .	41
7.3	The name of logic contracts in Table 7.1 and Table 7.2 with source code. .	42
7.4	The function prototypes of logic contracts in Table 7.1 and Table 7.2 with- out source code. . . . .	44
7.5	The number of distinct logic contract addresses in two datasets. . . . .	45
7.6	The top 5 proxy contracts with the same bytecode and their contract name. (part 1) . . . . .	46
7.7	The top 5 proxy contracts with the same bytecode and their contract name. (part 2) . . . . .	46
7.8	Proxy contract names and their logic contract addresses. . . . .	47
7.9	Upgrade events. . . . .	48



# List of Algorithms

1	Binary search pseudocode . . . . .	31
---	------------------------------------	----

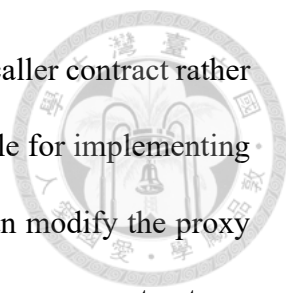


# Chapter 1 Introduction

Decentralized Finance (DeFi) allows everyone to deploy smart contracts on a public blockchain, such as Ethereum, for providing various financial services. As smart contracts are code written by human developers, they may contain bugs. Many of these bugs have been exploited and have caused DeFi users and developers significant money loss. For example, in 2016, the DAO attack [9] exploited a reentrancy vulnerability [20] in a smart contract to steal the equivalent of \$70 million. Unfortunately, unlike directly changing code when patching other software systems, developers cannot fix buggy code in a deployed smart contract because of the immutable feature of blockchains.

The concept of the proxy contract has emerged recently as a remedy to immutable buggy contracts. By putting the actual program logic into a secondary contract called a logic contract, a proxy contract can “upgrade” by switching to a different logic contract. In other words, a proxy contract can upgrade its program logic by modifying its storage (which stores the address of the logic contract) without modifying its code.

In this work, we focus on investigating the proxy contracts on Ethereum because it is the most popular public blockchain (by market capitalization) supporting smart contract functionality. In Ethereum, proxy contracts can be implemented using the `delegatecall` opcode. The `delegatecall` opcode is similar to `call`, but with one distinction: when called



using the delegatecall opcode, the function runs in the context of the caller contract rather than the callee contract.<sup>1</sup> Therefore, the delegatecall opcode is suitable for implementing proxy contracts because the function in the logic (callee) contract can modify the proxy (caller) contract's Ethereum virtual machine (EVM) storage, and the proxy contract can update to a new logic contract while preserving its storage information. Moreover, a logic contract can be used by multiple proxy contracts simultaneously without interfering with each other's storage.

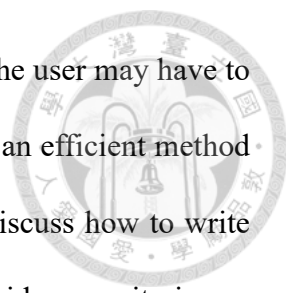
On the one hand, using proxy contracts enables upgradability and fixes smart contract bugs. On the other hand, misuse of proxy contracts may introduce new attack vectors. Particularly, when implemented using Ethereum's delegatecall opcode, a proxy contract may encounter function collisions and storage collisions, whose exploitation can deceive users into invoking unintended code or manipulate the EVM storage. For example, an attacker[4] exploited storage collisions and stole \$1.1 millions of AUDIO [3] tokens. This attack is listed in Rekt [30], a website listing all blockchain attacks. Briefly speaking, a function collision occurs when the proxy contract contains a function whose function signature<sup>2</sup> is the same as a function in the logic contract. A storage collision occurs when variables of proxy contracts or logic contracts conflict. It is because both proxy contracts and their logic contracts use the same storage. Thus, it is imperative to investigate the prevalence of proxy contracts and whether they are vulnerable.

However, to our knowledge, no prior studies have thoroughly analyzed proxy contracts' prevalence and whether they are vulnerable to function collisions and storage collisions. The tool Slither [16] can detect function and storage collisions if users specify the proxy contract and all its corresponding logic contracts. However, it is challenging

---

<sup>1</sup>By contrast, when called using the call opcode, the function runs in the context of the callee contract.

<sup>2</sup>A function signature is the first four bytes of the hash value of a function prototype.



for users to identify all previous logic contracts of a proxy contract; the user may have to examine all transactions so far. In contrast, in this work, we propose an efficient method to automate this search of logic contracts. Several works [19, 38] discuss how to write a proxy contract and maintain its storage. However, they do not consider security issues caused by the proxy contract's upgradability. Worse yet, many did not provide a concrete definition of proxy contracts, making it difficult to reproduce and compare against their results.

To address these limitations of prior studies, in this work, we first survey related EIPs and formulate an actionable definition of a proxy contract. We define a proxy contract as a contract whose fallback function uses the `delegatecall` opcode to forward the received `calldata`<sup>3</sup> to a logic contract.

Based on this definition, we develop a tool called ProxyChecker to determine (1) whether a contract is a proxy contract and (2) whether the proxy contract is safe to use. One can use ProxyChecker by just giving a smart contract address without any other input so it is easy to use. ProxyChecker contains three components, Delegatecall Detector, Dynamic Analyzer, and Contract Checker. Delegatecall Detector checks if a contract contains any `delegatecall` and passes the result to Dynamic Analyzer. Dynamic Analyzer emulates an EVM with a custom input to check if the contract meets our proxy contract definition. To implement Dynamic Analyzer, we extend an open source project, Octopus [28], to support the latest opcodes. Given a detected proxy contract, Contract Checker runs an efficient method for crawling all its logic contracts. It then crawls the source code of the proxy contract and logic contracts and leverages Slither for identifying potential function or storage collisions. We also discuss several implementation challenges, such as preventing com-

---

<sup>3</sup>A `calldata` is an input of a transaction. It contains the function signature and parameters of the function to be executed.

pilation errors and ensuring a consistent inheritance order when a contract inherits from multiple contracts.



Our research questions are the following:

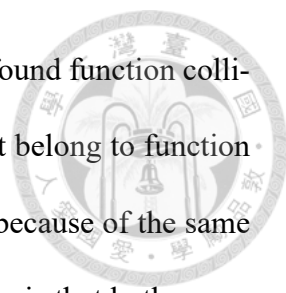
1. How widely are proxy contracts used?
2. What are the common usages of logic contracts?
3. What are the common usages of proxy contracts?
4. How frequently does a proxy contract upgrade?
5. How many proxy contracts contain function collisions or storage collisions?

To answer these questions, we sampled contracts from the transactions within six block ranges: 100000 blocks each February between 2017 to 2022. We analyzed the sampled contracts using ProxyChecker and found that the percentage of proxy contracts increased from 1% in 2017 to 88% in 2022. The increase is reasonable as proxy contracts were first proposed in 2018 and refined by several EIPs later. Our manual inspection suggests that the increase is also due to the NFT trend. As for the proxy contract type and its relationship with logic contracts, our analysis revealed that the most common type of proxy contract is the minimal proxy contract<sup>4</sup>, and most proxy contracts are pointing to a wallet contract [10]. Also, a logic contract may be pointed by several proxy contracts, but these proxy contracts usually have the same bytcodes. We also analyzed the upgrade events in proxy contracts and found that only 1139 proxy contracts have been upgraded for fixing bugs or supporting new features, and the average blocks between upgrades in

---

<sup>4</sup>The term is introduced in the 1167th Ethereum Improvement Proposals (EIP-1167).





each dataset range from 33529 to 11895666. For security issues, we found function collisions are easy to create, and most of the security issues in our dataset belong to function collisions. Moreover, all of the function collisions in our dataset are because of the same function name in both proxy contracts and logic contracts. The reason is that both proxy contracts and logic contracts are inherited from the same smart contracts for managing storage in EVM. Besides, we found that storage collisions are easy to exploit by the proxy owner.

We provide some suggestions for users and developers to avoid triggering the two main security issues. We also show several suggested usages of proxy contracts and their advantages and disadvantages. We conclude that the best usage of proxy contracts is for personal usage, like creating one's proxy contract for a wallet contract. As a result, there is no need to worry about malicious proxy owners. This usage is also the primary usage in our dataset.

Our main contributions are the following:

- Build an easy-to-use ProxyChecker.
- Present a systematic result of proxy contracts with three block ranges.
- Provide security suggestions for users and developers.
- Give some suggested usages of proxy contracts.
- We show that a proxy owner can manipulate the proxy contract storage.



## Chapter 2 Background

Since the `delegatecall` opcode and fallback function are the core of a proxy contract, we need to know how opcodes and fallback function work in Ethereum virtual machine (EVM). Moreover, the two main security issues of a proxy contract, namely, function collision and storage collision, are related to the storage layout and the function signature of a contract. We will explain these in the following sections.

### 2.1 EVM

EVM is a stack machine that executes sequences of bytecodes. The main components in the EVM are shown in Figure 2.1, and we will explain each of them in turn below.

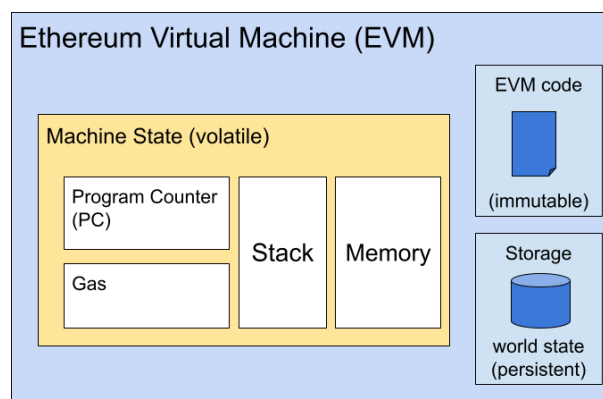
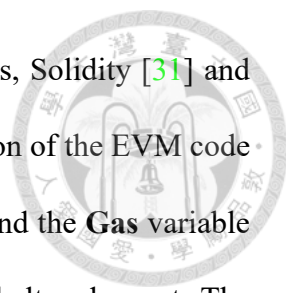


Figure 2.1: EVM components

The **EVM code** is the bytecode of a smart contract and is stored on the Ethereum



blockchain. The bytecodes are usually compiled from two compilers, Solidity [31] and Vyper [35]. The **program counter (PC)** variable indicates the location of the EVM code that the EVM is executing. Executing the EVM code costs the gas and the **Gas** variable stores the remaining gas. If the EVM runs out of gas, the EVM will halt and revert. The **stack** contains at most 1024 items, each of which is 256 bits. The **memory** is a word-addressed byte array. Each smart contract maintains its **storage** to store data, and the storage can theoretically contain up to  $2^{256}$  slots if we ignore the cost of gas. Each storage slot is 256 bits. All smart contract storage is persistent in the Ethereum blockchain. At the beginning of the EVM execution, both stack and memory are empty, but the storage may contain data from previous transactions.

## 2.2 EVM Opcode

The opcodes to the EVM [13] are the same as instructions to the CPU. The program code of a smart contract is called bytecode. It is composed of a bunch of opcodes that smart contracts will execute. Since the EVM is executed on the blockchain, users may need to pay the miner different amounts of gas to execute different opcodes.

Proxy contracts are based on the `delegatecall` opcode. The `delegatecall` opcode can execute functions of other contracts but modify the storage in proxy contracts. This is the root cause of storage collisions mentioned in Section 4.



## 2.3 EVM Storage Layout

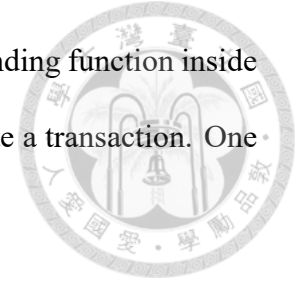
In EVM, each global variable, called state variable, defined in a smart contract has its storage slot. The order of declaration of the variables is the same as the index of that variable in the storage. For example, the first global variable in a contract is stored at storage slot 0, while the second global variable is stored at storage slot 1. However, different types of global variables may have different storage layouts [31]. For some base types of global variables, such as `uint`, the variable's value is stored at its storage slot. For dynamic array or `string`, the value stored at its storage slot is the size of the array, and the storage location of the array is at storage slot  $\text{keccak256}(\text{index of array in the contract}) + \text{offset}$ . For mapping, there is no value stored in its storage slot, the storage location of the data in the mapping is at storage slot  $\text{keccak256}(\text{the key of the data, index of mapping in the contract})$ .

## 2.4 Function Signature

In Ethereum, a function is identified by its signature, which is the first four bytes from hashing the prototype string of a function using the `keccak256` hash function. A prototype string of a function is a string concatenating the function name and the type of function parameters. For example, the prototype string of a function `test(address a, address b)` is `test(address,address)`, and the value of hashing `test(address,address)` is `0x2b6d0cebcc14bc50c4c35dd046aed41466c895d3f0b39f6f60d1f2fd199c5ad3`. Thus, the function signature of the function `test(address a, address b)` is `0x2b6d0ceb`.

If the receiver of a transaction is a smart contract, the EVM will treat the first four

bytes of the calldata as a function signature and execute the corresponding function inside the smart contract. The calldata is a byte-addressable data field inside a transaction. One can send any calldata along with a transaction.



## 2.5 Fallback Function

If the function signature within a calldata does not match any function in a contract, then the contract will execute the fallback function as the default. The name of the fallback function may be different in different compilers. For example, the fallback function is called fallback function in Solidity but called default function in Vyper.

## 2.6 Ethereum Improvement Proposals

Ethereum Improvement Proposals (EIP) are the suggested improvements for developing a smart contract. Although EIPs can be proposed by anyone and may not be adopted by Ethereum officially, developers often regard EIPs as a reference when creating contracts.

We searched among existing EIPs using the keyword “proxy” and found that the ideas of proxy contracts have been proposed or discussed in EIP-897, EIP-1167, EIP-1822, EIP-1538, EIP-1967, and EIP-2535. All of them used the term proxy contract, but only EIP-1822 defined a proxy contract: “The contract A which stores data, but uses the logic of external contract B by way of `delegatecall()`”.

As we will explain later, the definition in EIP-1822 is loose and may include contracts that use `delegatecall` for other purposes, such as making calls to library functions. We will

present a precise definition suitable for measurements and security analysis in Section 4.

EIP-897 (titled “DelegateProxy”) is the first EIP touching upon the concept of proxy contracts and trying to standardize the proxy contract interface. However, there is no definition or explanation of what a proxy contract is.

EIP-1167 (titled “Minimal Proxy Contract”) aims to minimize the cost of deploying a contract that was deployed before. What a minimal proxy contract does is using delegatecall to call another contract with all calldata the minimal proxy contract received. The cost of deploying a smart contract is the size of the code, and minimal proxy contract only contains indispensable opcodes for using delegatecall instead of the same smart contract. Thus, the cost of deploying a minimal proxy contract is much less than deploying a copy of a smart contract but their functionality are the same. In EIP-1167, the length of a proxy contract’s bytecode is usually less than 100 bytes, and the logic contract address is hardcoded in the bytecode of proxy contract. However, EIP-1167 does not provide a clear definition of the minimal proxy contract.

EIP-1822 (titled “Upgradeable Proxy Standard”) proposes a standard upgradeable proxy contract that can be compatible with all contracts. It uses a specific storage slot at the value of keccak256(“PROXIABLE”) to store the logic contract address. EIP-1822 contains terminology “Proxy Contract” and gives the definition. However, this definition means any contract using delegatecall is a proxy contract. We argue against this definition in Section 4.

EIP-1538 (titled “Transparent Contract Standard”) is replaced by EIP-1967 (titled “Standard Proxy Storage Slots”) by the same proposer. The main goal of EIP-1967 is to save the specific data in a specific storage slot to avoid storage collisions. The logic con-

tract address is saved at storage slot at the value of  $\text{keccak256}(\text{"eip1967.proxy.implementation"}) - 1$ .



It also defines a special form of logic contract called beacon contract, which is usually used for keeping the logic address for multiple proxy contracts in a single location. The beacon contract address is saved at the storage slot indexed by  $\text{keccak256}(\text{"eip1967.proxy.beacon"}) - 1$  of the proxy contract. The administrator of the proxy contract, which is the proxy owner, is saved at the storage slot indexed by  $\text{keccak256}(\text{"eip1967.proxy.admin"}) - 1$ .

The maximum size of a smart contract is 24KB, so every smart contract can only contain a limited number of functions. Nevertheless, EIP-2535 (titled "Diamonds, Multi-Facet Proxy") wants to make a proxy contract that can contain more functions regardless of the 24KB size limit. For simplicity, we will call this kind of proxy contract as diamond contract. A diamond contract uses the mapping type to map a function signature to a logic contract address. Theoretically, a diamond contract owner can register up to  $2^{256}$  number of function signatures to different logic contracts. The delegatecall will only be triggered if the function signature is registered in the proxy contract; otherwise, it will revert.



## Chapter 3 Related Work

### 3.1 Smart Contract Analysis Tools

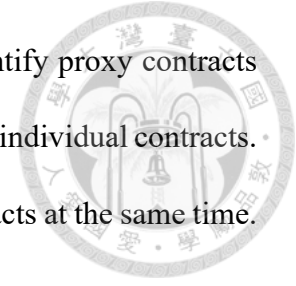
Although there are many smart contract analysis tools, to our knowledge, only Slither supports security checks on proxy contracts. Slither [16] is an open-source static analysis framework for smart contracts. It contains many detectors for detecting vulnerabilities. Among these detectors, one detector is for proxy contracts, called slither-check-upgradeability. This detector can help developers review proxy contracts to prevent bugs inside the smart contracts.

Though the detector can perform security analysis on proxy contracts, it needs the source code of both proxy contracts and their logic contracts. Users can extract the logic contract addresses from previous transactions. However, a smart contract may have thousands of previous transactions so it is hard for users to examine all these transactions. To address the problem, we provide a method to collect all of the logic contracts a proxy contract used before in our tool. We explain the details in Section 5.

Mythril [8], Manticore [21], rattle [34] and ConFuzzius [33] are smart contract analysis tools, but none of them include security check on proxy contracts. For example, rattle generates control-flow graphs of smart contracts, and Mythril performs security analysis



on smart contracts. However, it is hard to extend these tools to identify proxy contracts and detect their security issues because these tools focus on analyzing individual contracts. In contrast, we need to consider both proxy contracts and logic contracts at the same time.



## 3.2 Ethereum Virtual Machine Emulator

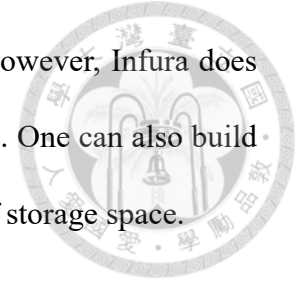
Based on the definition in Chapter 4.1, we need to know each state of the stack when executing the EVM to verify a smart contract is a proxy contract. Thus, an EVM emulator is needed and we create an EVM emulator component in our ProxyChecker. Several existing tools have similar functionality and may replace our emulator component but with some constraints. We briefly review these emulation tools in the following sections.

### 3.2.1 Go-Ethereum

Go-Ethereum [32] is an official implementation of the Ethereum protocol by Golang. One can use it to interact with the Ethereum blockchain. In Go-Ethereum, a command called `evm` can execute the bytecode and output the state of stack and memory at each opcode execution. This command is useful for finding a proxy contract because it can display the state of stack, memory, and storage of each opcode operation. We do not adopt Go-Ethereum in our tool because it requires information about the current Ethereum state, and the information is too large to be handled efficiently.

Go-Ethereum also provides JSON-RPC APIs. These APIs are used to connect to an Ethereum JSON-RPC node like Infura [18] or QuickNode[29]. One of the APIs is called `debug_tracecall`, which can emulate the transaction by providing input and returning all information of the transaction, such as every state of stack, memory, and storage of EVM.

This API can replace our EVM-emulator component of our tool. However, Infura does not support this API, and QuickNode needs to pay for using the API. One can also build a JSON-RPC node of Ethereum, but it costs a lot of time and a lot of storage space.



### 3.2.2 E-EVM

E-EVM[24] is a tool that emulates the bytecode of Ethereum and visualizes the control-flow graph with the state of the stack. Initially, we think the tool may help us to find proxy contracts. However, we cannot reproduce the result by the source code which is provided on Github. As a result, we decide not to choose this tool as our component.

## 3.3 Proxy Contract Related

Several studies [19, 38] focus on how to write a proxy contract and show different concepts of how to maintain the storage in a proxy contract to prevent the storage collision problem.

Etherscan [14] is a website that provides information about Ethereum, such as every transaction and contract's bytecode. It also keeps upgrading to provide more features. To our knowledge, only Etherscan supports the functionality of detecting proxy contracts. Proxy contract detection is one of their newly-introduced features. However, there is no document about it, so we do not know how it works. In comparison, in this work, we provide our definition of proxy contracts and how to detect them.



## Chapter 4 Security Issues

We first define what a proxy contract is, then discuss the two security issues of proxy contracts in this section.

### 4.1 Definition of Proxy Contracts

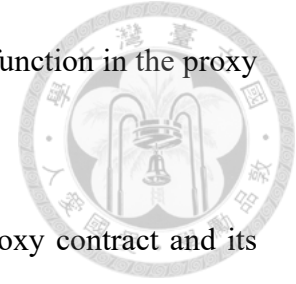
Since there is no general definition of proxy contracts, we first formulate our definition of proxy contracts. We deem a contract a proxy contract if it uses `delegatecall` in its fallback function to forward the `calldata` it received to another contract. The contract receiving the forwarded `calldata` is defined as a logic contract of the proxy contract.

Our definition is compatible with the usage of all known proxy-related EIPs. Moreover, the `calldata` forwarding condition excludes the use of calling a library contract. A library contract is a smart contract that contains reusable code for general usage. For example, `SafeMath` [26] is a library for overflow prevention during math calculation.

### 4.2 Function Collisions

A function collision is when a proxy contract contains some functions whose function signatures are the same as some functions in logic contracts. In this situation, a user may

expect to execute a function in a logic contract but execute another function in the proxy contract instead.



Code 4.1 is an example of a function collision. There is a proxy contract and its logic contract is contract A. The function signatures of the proxyOwner() function and the clash550254402() function are the same. Therefore, if a user wants to execute clash55025-4402(), he will send the function signature of clash550254402() to the proxy contract. However, the proxy contract will execute proxyOwner() instead of the fallback function because its function signature is matched, which is not what the user expected.

A malicious proxy contract may leverage function collisions to deceive the user into executing the malicious code. Besides, it is easy to find two functions with different names with the same function signature if one does not restrict the function name's format and the function signature's value. For example, creating two different functions with the same function signature takes around 1 second on a laptop with an i7-7700HQ CPU. It is slightly difficult to find a function whose function signature is a specific value, and it takes several hours to find one on the same laptop, but it is arguably affordable for a dedicated, monetary-driven attacker.

```
1 pragma solidity ^0.4.23;
2
3 contract Proxy {
4     function proxyOwner() public returns (address) {
5         return msg.sender;
6     }
7     function implementation() public view returns (address);
8     function () payable public {
9         address _impl = implementation();
10
11     assembly {
12         calldatacopy(0, 0, calldatasize)
```



```

13         let result := delegatecall(gas, _impl, 0, calldatasize, 0,
14             0)
15         returndatacopy(0, 0, returndatasize)
16
17         switch result
18         case 0 { revert(0, returndatasize) }
19         default { return(0, returndatasize) }
20     }
21 }
22
23 contract A {
24     function clash550254402() public view returns (address){
25         return address(this);
26     }
27     function test() public view returns (address){
28         return msg.sender;
29     }
30 }

```

Listing 4.1: Function collision example

It is worth noting that some design patterns for storage management mentioned in [38] will result in function collisions. In these design patterns, both the proxy and logic contracts inherit from a contract that stores all variables. The contract may contain some functions for accessing the variables. However, it causes a function collision because both proxy contracts and logic contracts contain the same function. We do not consider this a false positive of function collisions because users will not know whether the logic in the functions is the same unless checking the source code.



### 4.3 Storage Collisions

A storage collision is an issue in storage slots. Since a proxy contract uses the `delegatecall` opcode to forward `calldata` to a logic contract, the logic contract can modify the storage of the proxy contract. We know the variable is stored at the index of the storage slot according to the variables' order. However, if the variables' order in the logic contract differs from the proxy contract, some unexpected results may occur.

Code 4.2 is an example of a storage collision. If users want to execute the `add()` function in a logic contract by calling a proxy contract, they may expect the return value is 3 because the "b+c" is written in the function. However, the return value is 1 because the `add()` function returns the sum of values in storage slot 0 and storage slot 1. Thus, the `add()` function will add the first variable and second variable in the smart contract, which are `a` and `b`, and return.

```
1 contract proxy{
2     uint a = 0;
3     uint b = 1;
4     uint c = 2;
5
6     function (){
7         assembly {
8             let result := delegatecall(gas(), "logic's address", 0,
9                 calldatasize(), 0, 0)
10            returndatacopy(0, 0, returndatasize())
11            switch result
12            case 0 { revert(0, returndatasize()) }
13            default { return(0, returndatasize()) }
14        }
15 }
```



```
16
17 contract logic{
18     uint b;
19     uint c;
20
21     function add() public returns (uint){
22         return b+c;
23     }
24 }
```

Listing 4.2: Storage collisions example 1

A storage collision will also happen at upgrading a proxy contract to a new logic contract. If the order of variables in the new logic contract differs from the old logic contract, values in those variables may be different.

Code 4.3 is another example of a storage collision. Usually, a logic contract contains an `initialize()` function to initialize the proxy contract's storage. The `initialize()` function is expected to be called only once after the proxy contract is created. By checking the value of initialized variable, we can prevent the function from executing again. However, if an initialized proxy contract is upgraded to the new logic contract called `newInitialOnce`, the `initialize()` function can be executed again because the value of the initialized variable is changed to `False`. Thus, it is important to ensure the variables' order should be the same between the old logic contract and the new logic contract when developers upgrade a proxy contract.

```
1 contract InitialOnce{
2     bool private initialized;
3     bool private boolean;
4
5     function initialize(){
6         require(!initialized);
7         boolean = false;
```

```

8     initialized = true;
9     }
10 }
11
12 contract newInitialOnce{
13     bool private boolean;
14     bool private initialized;
15
16     function initialize(){
17         require(!initialized);
18         boolean = false;
19         initialized = true;
20     }
21 }

```



Listing 4.3: Storage collisions example 2

Nevertheless, if a proxy contract is upgradable and has a storage collision, a malicious proxy contract owner can deliberately manipulate the storage of the proxy contract. For example, a new logic contract may contain a function that can change any storage slot to any value. As a result, if a token smart contract, such as YLD token [37], is deployed by an upgradable proxy contract, the owner of the proxy contract can manipulate the number of tokens or the ownership of the token by upgrading to a malicious logic contract that can modify any storage value.





# Chapter 5 PROXYCHECKER:

## Design and Implementation

This section presents the design and implementation of our tool, ProxyChecker. ProxyChecker is a tool for checking whether a smart contract address is a proxy contract or not. If it is a proxy contract, ProxyChecker will try to get the source code of both the proxy contract and its logic contracts. After that, it will do a security analysis on those with source code to check if there are any security issues and output the result.

It is worth noting that our ProxyChecker implementation currently supports analysis on Ethereum, but our idea and design apply to other smart contract platforms.

### 5.1 Overview

ProxyChecker contains three components, Delegatecall Detector, Dynamic Analyzer, and Contract Checker. The purpose of Delegatecall Detector is to filter out smart contracts containing no delegatecalls. The purpose of Dynamic Analyzer is to verify whether the calldata is forwarded by delegatecall in the fallback function. The purpose of Contract Checker is to check if there are any function collisions or storage collisions in a proxy contract.

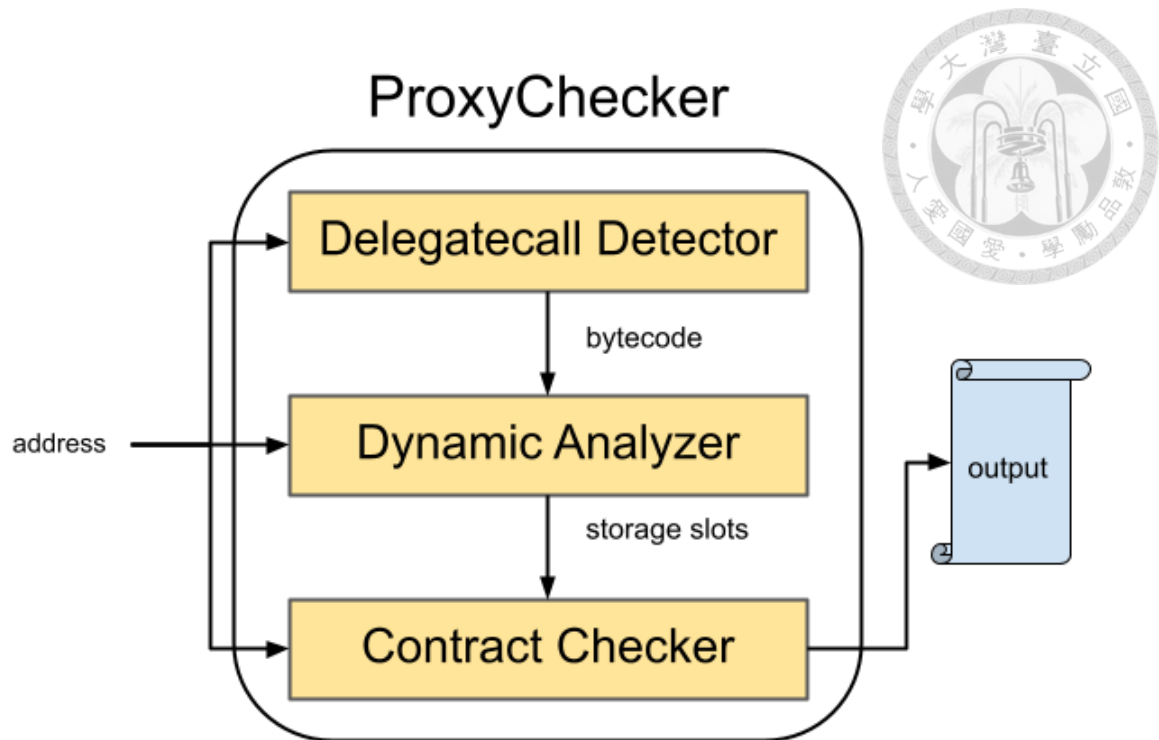


Figure 5.1: ProxyChecker overview

ProxyChecker only needs a smart contract address as input. First, Delegatecall Detector will check whether it is a contract and if there are any delegatecall in the smart contract. Then, it will forward contracts containing delegatecalls to Dynamic Analyzer. Dynamic Analyzer will construct a calldata and emulate the execution of the smart contract.

After the emulation, Dynamic Analyzer can verify whether the smart contract is a proxy contract or not. If the smart contract is a proxy contract, Dynamic Analyzer will pass the storage slot which stores the logic contract address to Contract Checker. Contract Checker will check the historical values of this storage slot and take these values as old logic contracts' addresses. Then, it will crawl the source code of the proxy contract and logic contracts from Etherscan, and merge them into a single file for the input of Slither. Last, we use slither-check-upgradeability from Slither to check if there are any function collisions or storage collisions. We will explain all the components in the following sec-

tions.

While developing ProxyChecker, we met the following challenges:

1. Some opcodes depend on the current state of Ethereum but the state of Ethereum changes at every new block creation.
2. Old logic contracts can be found in transactions but the number of transactions may be too large to search one by one.
3. Because of inheritance, we cannot simply merge multiple source code files into one file in any order.

To deal with the first challenge, we use the latest value or a reasonable fixed value for these opcodes. For the second challenge, we propose a binary search method to find, which will be explained later. For the third challenge, we get the inheritance of each contract before merging them, or it will cause a compilation error.

## 5.2 Delegatecall Detector

Fig. 5.2 shows the workflow of Delegatecall Detector.

Since the key opcode of a proxy contract is the delegatecall opcode, if a smart contract does not contain any delegatecall opcodes, it must not be a proxy contract. Thus, we need a smart contract disassembler, which can decompile Ethereum bytecode into a sequence of opcodes. We use the disassembler from the open-source project Octopus [28] to extract the opcode from the bytecode of the contract and check if it contains any delegatecall opcodes. We only proceed to the next component if it contains any delegatecall opcodes.



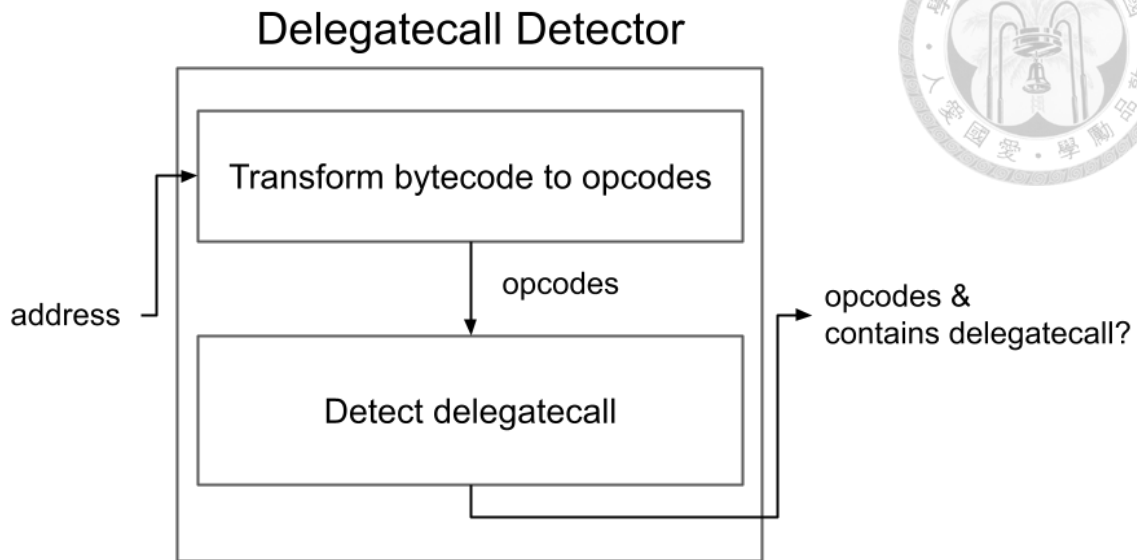


Figure 5.2: Delegatecall Detector

Otherwise, we stop ProxyChecker.

### 5.3 Dynamic Analyzer

Fig. 5.3 shows the workflow of Dynamic Analyzer. Though delegatecall opcodes are in the contract, they may in other functions rather than in the fallback function. Even if a contract contains delegatecall opcodes in the fallback function, it is still possible that it is not a proxy contract but for making library calls or other purposes. Therefore, we need to get the parameters of the delegatecall when executing the fallback function to ensure the smart contract meets our definition of the proxy contract.

Dynamic Analyzer is based on Octopus and it will do the following step by step.

1. Construct a calldata to trigger the fallback function.
2. Dynamically analyze and extract the parameters.

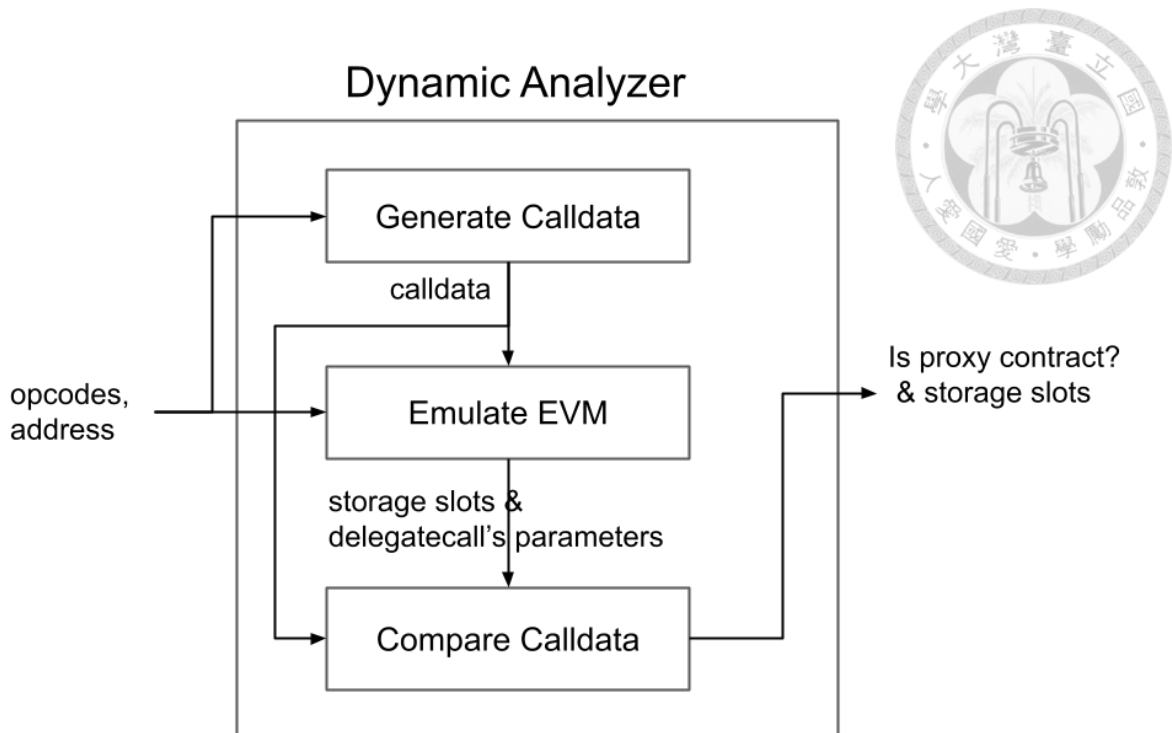


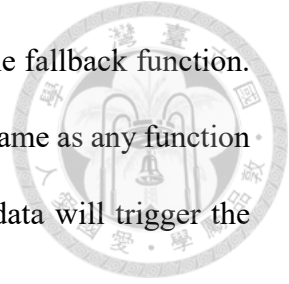
Figure 5.3: Dynamic Analyzer

3. Compare whether the calldata is passed to the fallback function

### 5.3.1 Create a Calldata

To check whether a contract matches our definition of proxy contracts, we first construct a calldata such that the execution of a smart contract will trigger the fallback function. The contract will execute its fallback function if the first four bytes of calldata do not match any function signature of functions. Thus, for making a smart contract to execute the fallback function, we can leverage this property by creating a custom calldata. Furthermore, according to the supplement documentation of [7], both Solidity and Vyper will compile all function signatures following a PUSH4 opcode. Thus, we extract all 4 bytes of data following a PUSH4 opcode as function signatures and avoid using them as our custom calldata. These function signatures may contain some false positives, such as some four-byte constant variables defined in the smart contract. However, these false pos-

itives will not affect our goal of making a smart contract to trigger the fallback function. As a result, we construct a calldata whose first four bytes are not the same as any function signatures extracted from a contract, thereby ensuring that this calldata will trigger the contract's fallback function.



### 5.3.2 Dynamically Analyze and Extract the Parameters

In the second step, we dynamically analyze the smart contract with the calldata from step 1. We need to get the calldata passed by delegatecalls so we can compare it with the calldata created from step 1 in step 3. The calldata is one of the parameters of delegatecalls. Since EVM is a stack machine, all opcode parameters are stored in the EVM stack. Moreover, the stack is also affected by the EVM's memory and storage so we emulate the EVM to collect the parameters.

Due to some limitations of the existing emulator tool as we mentioned in Section 3, we build our EVM emulator component based on Octopus. Octopus is an open-source security analysis framework for WebAssembly and smart contracts. Moreover, it also supports different types of smart contracts, such as Ethereum smart contracts, EOS [11] smart contracts, and NEO [23] smart contracts. For Ethereum smart contracts, Octopus provides many modules to analyze the bytecode of Ethereum smart contracts. Although the project's last update was two years ago, and Ethereum had introduced more opcodes, we chose it as our base because the framework of Octopus is easy to extend. Since the components of Octopus are too many, we only focus on the components related to Ethereum and add some functionalities for our needs.

We modified the source code of Octopus to support the new opcodes introduced after

the last update of Octopus. We also add a Python [1] list as the stack, a Python bytes array as the memory, and a Python dictionary as the storage. Moreover, before we start the analysis, we need to set some parameters, such as the caller address and the contract address. We use a Python dictionary to record these parameters before emulation.

Octopus does not handle some opcodes whose values depend on the state of the global environment, such as the number opcode, which pushes the latest block number of Ethereum into the stack. We call them block opcodes because they depend on the block state of Ethereum. Our goal is to emulate a transaction and a transaction can be initiated by anyone at any time. Thus, for these block opcodes, we use possible values to emulate. All values are set by checking the latest value of Ethereum or a reasonable fixed value, so all of them are possible values that may be used in some transactions. The value of coinbase can be replaced by another value because anyone can be the miner of a block. The value of basefee can also be replaced by any other value, but it is more likely between the highest value and lowest value in history. However, this method still adds some uncertainty to our result. For example, a smart contract may be designed to execute only under a specific miner by leveraging coinbase opcode. Thus, if we meet these kinds of opcodes in our analysis, we put an uncertainty label in the result to notify the users. In our dataset, none of the contracts contains these kinds of opcodes and delegatecall at the same time, which means it does not affect our result. These block opcodes and values are listed in Table 5.1.

For some opcodes like call or delegatecall, which Octopus does not implement, they need to call another contract and get the execution result to push into the stack. To get the result of these kinds of opcodes, we create another EVM-emulator instance and execute it. After that, we put the result back into the stack of the original EVM-emulator



Table 5.1: Values for block opcodes

<b>opcode</b>	<b>value</b>
blockhash	latest value of block
difficulty	latest value of block
chainid	1
gaslimit	latest value of block
basefee	50000000000
timestamp	latest value of block
coinbase	0xbbbbbbbbbbbbbbbbbbbb bbbbbbbbbbbbbbbbbbbbbb
number	latest value of block
gasprice	latest value of block

to keep emulating. For some opcodes like CREATE or CREATE2, they will put the bytecode on Ethereum at a smart contract address. Since we cannot know the exact contract address of the newly created one, we use a fixed address instead. If the emulator encounters this address, we treat it like a normal smart contract. The method is acceptable because the probability of address collision in Ethereum is low. Besides, no smart contract encounters the CREATE or CREATE2 opcode during emulation in our dataset.

After the emulation, if we encounter any delegatecall, we pass the calldata it brings and the information of the contract storage slot to step 3. Otherwise, we halt the Proxy-Checker.

### 5.3.3 Compare the Calldata

By our definition of a proxy contract, a proxy contract must forward the same calldata it received to a logic contract. Thus, we need to compare the calldata we constructed in step 1 and the parameters of delegatecall in step 2. If they are the same, then the contract is a proxy contract. Otherwise, the contract is not a proxy contract but a smart contract doing a library call.





## 5.4 Contract Checker

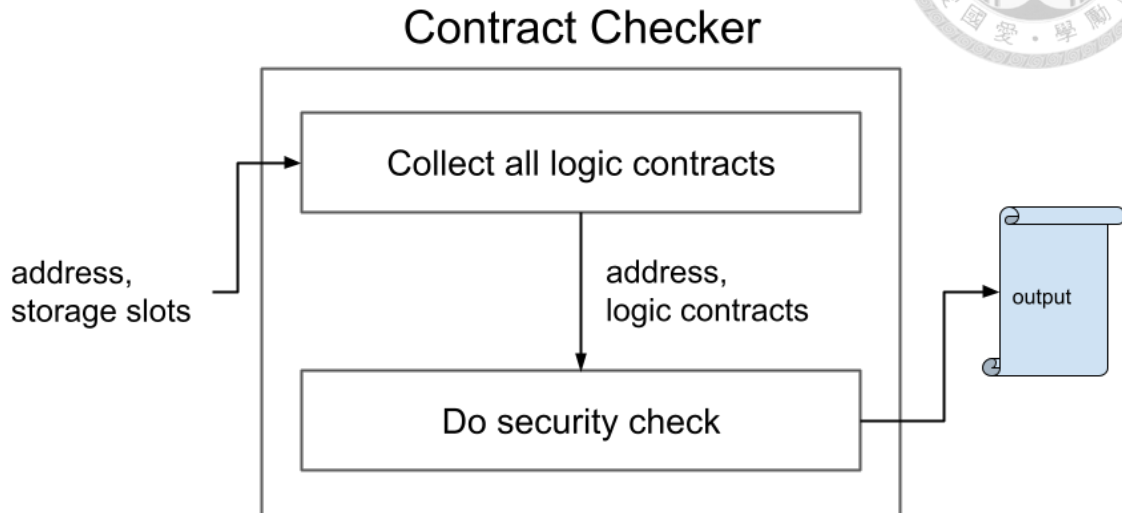


Figure 5.4: Contract Checker

Fig. 5.4 shows the workflow of Contract Checker. Contract Checker is for checking the security of proxy contracts. Our current implementation focuses on detecting function and storage collisions, as introduced in Section 4.

Moreover, we want to check whether the source code of contracts are provided because users should only interact with smart contracts with source code provided, or they cannot know what the contracts will do.

From Section 4, we know that all logic contracts used by the proxy contract before may affect the security of the proxy contract. As a result, we need to crawl all the logic contracts used by the proxy contract.

First, we try to find where the address of the logic contract is stored. The address of the logic contract must be stored in the following three places:

1. bytecode of the proxy contract.

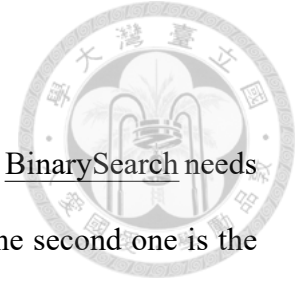
2. specific storage slot.
3. another smart contract.



We first check if the address of the logic contract matches the value of any storage slot we stored in step 2. If they are matched, the storage slot is where the logic contract is stored. Otherwise, we check if the logic contract address appears in the bytecode of the proxy contract. If the logic contract address does not appear in the bytecode of the proxy contract, the logic contract address is stored in another contract.

Suppose the logic contract address is stored in the bytecode of the proxy contract. In that case, this logic contract is the only one the proxy contract uses because it is impossible to change the deployed bytecode. If the logic contract address is stored at a specific storage slot, then all values stored in this storage slot before are the all logic contract addresses used by the proxy contract. However, in Ethereum, there is no API for getting all values that had been stored in a storage slot but an API called [web3.eth.getStorageAt](#) to get the value of a storage slot of an address at a specific block number. A naive solution is to access a storage slot from block 1 to the latest block. However, the current block number is more than 14000000, so it needs to call the [web3.eth.getStorageAt](#) API more than 14000000 times for a proxy contract, which is not practical. Thus, we apply a binary search to locate all logic contract addresses from block 1 to block 14800000 (the latest block number during our experiment). This method is based on the assumption that proxy contracts would not use the same logic contract twice. We consider it acceptable because an upgradable proxy contract is upgraded if the old logic contract contains loopholes or some deprecated functions. It is unreasonable to upgrade to an old logic contract. Nevertheless, by this method, we only need to call the [web3.eth.getStorageAt](#) API 26 times per proxy contract

on average rather than 14800000 times in the naive method.



Algorithm 1 is the pseudocode of the binary search. The function `BinarySearch` needs four parameters, the first one is the left bound of the block range, the second one is the right bound of the block range, the third one is a set storing all logic contract addresses, and the last one is the storage slot for the logic contract address. The algorithm first checks the values in the slot of the left and right bounds of the block range. If the values are different, we check the value of the storage slot from the middle of the block range. If the left and middle ones are different, it calls itself again with the range  $l$  to  $mid$ . Similar to the middle and the right bound. Last, the function returns  $S$ , which contains all the logic contract addresses.

---

**ALGORITHM 1**

Binary search pseudocode

---

$l, r$  are the left and right boundary of a block range.

$S$  is a set for storing value we get from the `getStorageAt`.

slot is the storage slot position.

```

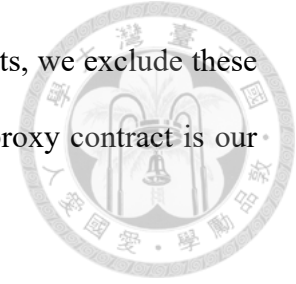
function BINARYSEARCH( $l, r, S, slot$ )
   $l_v \leftarrow getStorageAt(slot, l)$ 
   $r_v \leftarrow getStorageAt(slot, r)$ 
   $S \leftarrow S \cup \{l_v, r_v\}$ 
  if  $l_v \neq r_v$  then
     $mid \leftarrow (l + r) / 2$ 
     $mid_v \leftarrow getStorageAt(slot, mid)$ 
    if  $l_v \neq mid_v$  then
       $S \leftarrow S \cup BinarySearch(l, mid, S, slot)$ 
    end if
    if  $mid_v \neq r_v$  then
       $S \leftarrow S \cup BinarySearch(mid + 1, r, S, slot)$ 
    end if
  end if
  return  $S$ 
end function

```

---

If the logic contract address is stored at another contract, we currently are not able to get all logic contracts the proxy contract used before. The reason is that the contract which stored the logic contract address may be changed, and for different contracts, the ways they

stored the logic contract address may be different. In our experiments, we exclude these proxy contracts for further analysis. The solution for this kind of proxy contract is our future work.



After collecting all logic contracts, we check if the source codes of the proxy contract and logic contracts are provided. Since Etherscan is the most popular platform to get information about Ethereum, we chose it to check whether a smart contract is provided with the source code. We call these smart contracts verified contracts. If the proxy contract or one of its logic contracts is not a verified contract, we consider the proxy contract to be insecure. If all contracts are verified contracts, we check if there are any function or storage collisions issues. From Section 2, we know slither-check-upgradeability can check both issues with source code, so we need to get the source code first.

We use an API from Etherscan to get the source code of these verified contracts. However, the format of the source code return from the API is not consistent. Some return a dictionary format of source code, while others return an array of source code. Moreover, a smart contract may inherit from other smart contracts. Each smart contract may have a single source code file. To prevent compilation errors, we need to put these files into one file and ensure their order corresponds to the inheritance. We create our format parser to get the inheritance of all smart contracts. We first put the smart contracts that do not inherit from any contracts, then put other contracts one by one if their inherited contracts are put above. Last, we apply these contracts on the slither-check-upgradeability to check if there are any function or storage collisions.

The output of ProxyChecker includes whether the smart contract is a proxy contract, all logic contract addresses, all available source code and if there are any function colli-

sions or storage collisions.



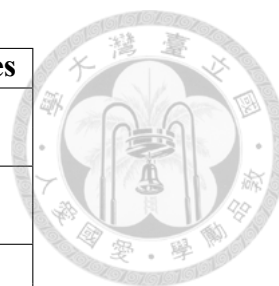


## Chapter 6 Data Collection and Analysis Result

We use ProxyChecker to examine smart contracts on the Ethereum mainnet. In this section, we report on our data collection and findings.

Due to the limited time and computing resources, we cannot analyze all smart contracts. Instead, we sample smart contracts from six block ranges to observe the trend of proxy contract usage. The resulting datasets are SC17, SC18, SC19, SC20, SC21, and SC22. Table 6.1 summarizes their block ranges, the date these blocks were added, and the number of contracts we sampled. Each block range has 100,000 blocks. Since there are too many smart contracts except for the first range, we sample at most two contracts from each block.

We need to collect the contract addresses from Ethereum mainnet. Fortunately, Google provides a BigQuery service for developers to query various kinds of data. One is a SQL database table that stores Ethereum mainnet data and updates to the latest block every few seconds in this service. We can query the information of smart contracts, such as the block number and block hash of the block that created this contract. We collected all the smart contract addresses for the block range 3140000 to 3240000 as SC17 because the number of smart contracts is only 6416.



	<b>Block range</b>	<b>Date</b>	<b># of samples</b>
<b>SC17</b>	3140000 3240000	Feb-07-2017 Feb-24-2017	6416
<b>SC18</b>	5040000 5140000	Feb-06-2018 Feb-23-2018	113281
<b>SC19</b>	7180000 7280000	Feb-05-2019 Feb-28-2019	91940
<b>SC20</b>	9430000 9530000	Feb-06-2020 Feb-22-2020	82218
<b>SC21</b>	11800000 11900000	Feb-06-2021 Feb-21-2021	91586
<b>SC22</b>	14160000 14260000	Feb-07-2022 Feb-23-2022	114301

Table 6.1: Dataset information

The numbers of contracts in the rest of the block ranges are more than 250000. For these block ranges, we fetch at most two contract addresses from each block for sampling because the number of smart contract addresses is too large. Since we are doing sampling, any two smart contracts from a block are acceptable. Thus, we pick the first two smart contract addresses from each block. We use the query shown in Fig. 6.1 to fetch at most two contract addresses from each block in the desired range. We collected 113281, 91940, 82218, 91586 and 114301 smart contract addresses from block range in 2018, 2019, 2020, 2021 and 2022 respectively. These datasets are denoted as SC18, SC19, SC20, SC21 and SC22. Some blocks may contain fewer than two smart contract addresses. However, we think it is acceptable because we are doing sampling in a period.

```
select *
FROM
(
  SELECT address,
  ARRAY_TO_STRING(function_sighashes, ','), block_number,
  ROW_NUMBER() OVER (PARTITION BY block_number) AS
row_n
FROM `bigquery-public-data.crypto-ethereum.contracts`
ORDER BY block_number DESC
)
WHERE row_n < 3
ORDER BY block_number DESC, row_n
```

Figure 6.1: The query we used in BigQuery

We apply our ProxyChecker to every contract address in the collected datasets.

Types	Dataset						
	SC17	SC18	SC19	SC20	SC21	SC22	
proxy	80 (1.25%)	1936 (1.71%)	1062 (1.16%)	13548 (16.48%)	31560 (34.46%)	101398 (88.71%)	
Not proxy	no delegatecall	6335 (98.74%)	110684 (97.7%)	90762 (98.72%)	68328 (83.1%)	59698 (65.18%)	12775 (11.18%)
	library call	0 (0%)	655 (0.58%)	106 (0.11%)	319 (0.39%)	309 (0.34%)	112 (0.1%)
	contract error	1 (0.01%)	6 (0.01%)	10 (0.01%)	23 (0.03%)	19 (0.02%)	16 (0.01%)
<b>Total</b>	6416	113281	91940	82218	91586	114301	

Table 6.2: Number and percentage of contracts of each type in different datasets.

Table 6.2 shows the result of applying the first two components, Delegatecall Detector and Dynamic Analyzer. The proxy type means the smart contract meets our proxy contract definition. Inside the not proxy type, we can classify them into three categories: no delegatecall, library call, and contract error. The no delegatecall type means the execution of the fallback function in a smart contract did not encounter any delegatecalls. The reason may be that the delegatecall is contained in other functions, or the smart contract does not contain any delegatecall. The library call type is the smart contract that executes the delegatecall in the fallback function, but the calldata we created is not forwarded into delegatecall. The contract error type is the smart contract that contains an invalid bytecode sequence such as popping from an empty stack or the smart contract that uses delegatecall or call opcode to call an invalid contract address such as address “0”. Besides, no smart contract encounters block opcodes mentioned in Table 5.1. Though it may be lost from sampling, it also means block opcodes are rare to be used in proxy contracts.

In dataset SC17, only 80 smart contracts belong to the proxy type, and most smart contracts belong to the no delegatecall type. It is reasonable because the proxy pattern had not been invented then. In datasets, SC18 and SC19, only around 1% of smart contracts are proxy contracts, which shows that proxy contracts were not popular at that time rather than other kinds of smart contracts. In dataset SC20, most smart contracts belong to the no delegatecall type either, but 13548 smart contracts belong to the proxy type, which



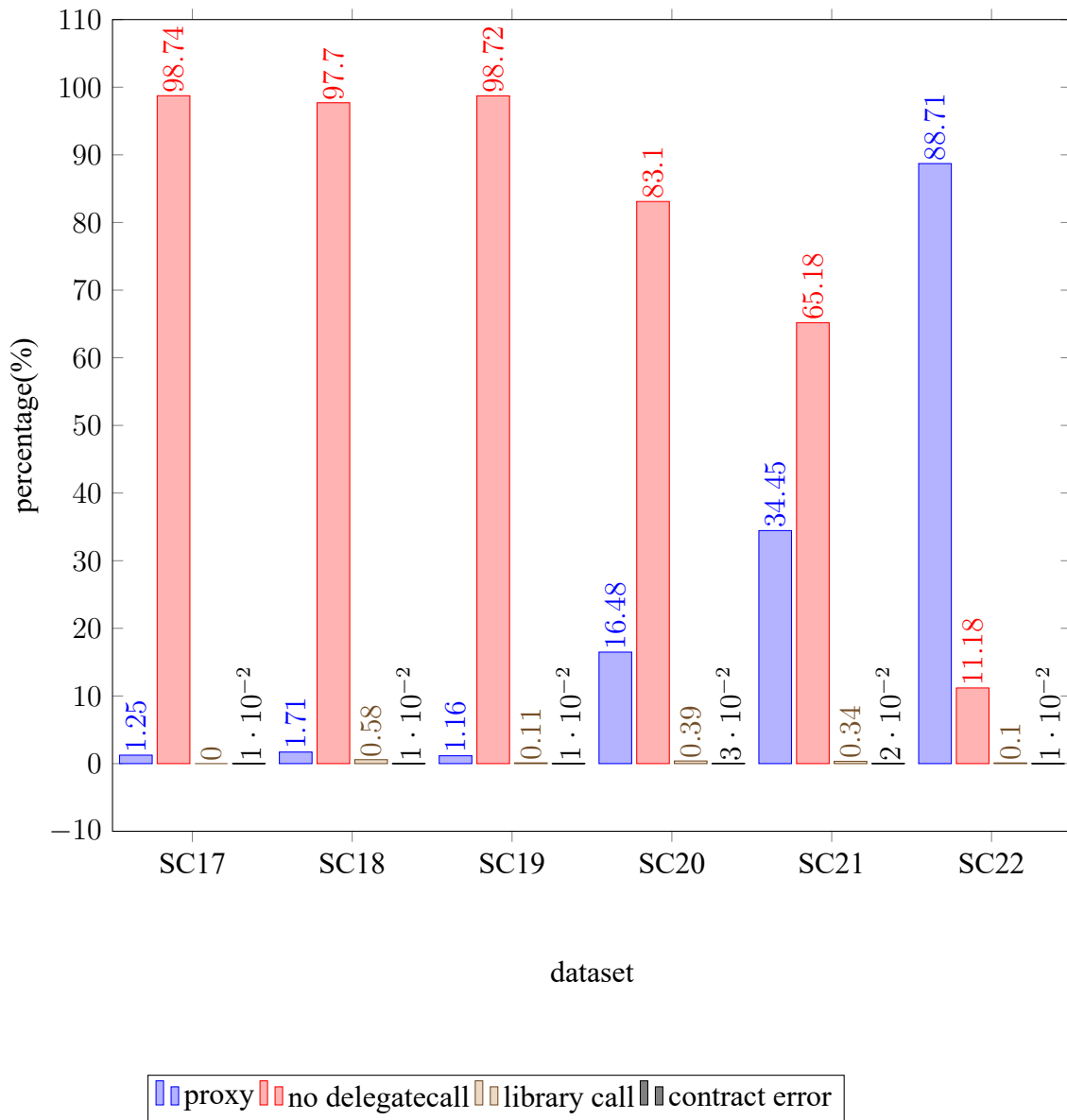


Figure 6.2: Result of Dynamic Analyzer

accounts for 16.48%. In dataset SC21, most smart contracts belong to the no delegatecall type either, but 31560 smart contracts belong to the proxy type, which accounts for 34.46%. In dataset SC22, most smart contracts belong to the proxy type, which accounts for 88.71%, and only 12775 contracts belong to the no delegatecall type, which accounts for 11%. Figure 6.2 shows the percentage histogram of each block range. From the figure, we notice that the use of proxy contracts are increasing from 2020 to 2022. The result answers our first research question.

For proxy contracts, we apply Contract Checker. After the first step of Contract Checker, which is collecting all logic contracts of the proxy contract, we can categorize these proxy contracts into different types of EIPs.

For those proxy contracts whose logic contract addresses are stored in bytecode, we classify them as EIP-1167 (minimal proxy contract). For those contracts whose logic contract addresses are stored at the keccak256(“PROXIABLE”) storage slot, we classify them as EIP-1822. For those contracts whose logic contract addresses are stored in keccak256(“eip1967.proxy.implementation”) storage slot, we classify them as EIP-1967. For those contracts whose logic contract addresses are not stored in bytecode or any storage slot, they must be stored in another smart contract, we give them a single type. For the rest of the proxy contracts whose logic contract addresses are stored in other storage slots, we classify them as the “others” type. The result is shown in Table 6.3.

<b>Types\Dataset</b>	<b>SC17</b>	<b>SC18</b>	<b>SC19</b>	<b>SC20</b>	<b>SC21</b>	<b>SC22</b>
EIP-1167	79	1924	317	7648	25802	51804
EIP-1822	0	0	0	0	207	0
EIP-1967	0	0	0	1068	876	244
another contract	0	7	4	909	418	1230
Others	1	5	741	3923	4257	48120

Table 6.3: The number of contracts classified by where logic contract address is stored

The two security issues we want to analyze will not happen on minimal proxy contracts because no variables or functions are defined in minimal proxy contracts. It also means minimal proxy contracts can not be upgraded. Currently, the “another contract” type is excluded because we do not know where the logic contract is stored. We will include it in our future work. Thus, we only move the type EIP-1822, EIP-1967, and others into the next stage of the security check, which is to get all logic contracts by the binary search method.

<b>Types\Dataset</b>	<b>SC17</b>	<b>SC18</b>	<b>SC19</b>	<b>SC20</b>	<b>SC21</b>	<b>SC22</b>
Verified	0	0	679	4398	5010	46890
Unverified	1	5	62	593	330	1474
(EIP-1822 & EIP-1967 & Others)	1	5	741	4991	5340	48364

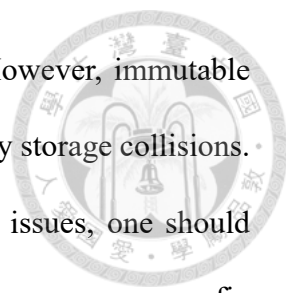
Table 6.4: Number of contracts is verified among EIP-1822, EIP-1967 and Others

After collecting the logic contract addresses of proxy contracts, we check whether both proxy contracts and logic contracts are verified contracts in the third component of ProxyChecker. Otherwise, we cannot apply the slither-check-upgradeability. Table 6.4 shows the number of proxy contracts and logic contracts that belong to verified contracts.

<b>Types\Dataset</b>	<b>SC17</b>	<b>SC18</b>	<b>SC19</b>	<b>SC20</b>	<b>SC21</b>	<b>SC22</b>
No collisions	0	0	102	2688	1086	900
function collisions	0	0	575	657	3835	44407
storage collisions	0	0	2	4	46	1269
function & storage collisions	0	0	0	1	28	13
error	0	0	0	1048	15	301
Verified	0	0	679	4398	5010	46890

Table 6.5: Number of contracts of collision issues among verified contracts

We apply the slither-check-upgradeability on these verified contracts and get the result shown in Table 6.5. The result answers our fifth research question. All function collision issues in the Table 6.5 are because of the same function name. They inherit from the same smart contract for maintaining storage layout, as we mentioned in Section 4. Many of the storage collisions may be false positives because the implementation of slither-check-



upgradeability recognizes immutable variables as state variables. However, immutable variables are stored in bytecode, not in storage, so it will not cause any storage collisions. Thus, if the output of ProxyChecker contains any storage collision issues, one should check if it is a false positive. Slither is an open-source project so anyone can propose a fix to it. Thus, fixing the false positive is our future work. The “error” type is because of the implementation of the switching compiler in slither-check-upgradeability. If the Solidity version differs between the proxy contract and logic contract, the error will occur during executing slither-check-upgradeability. The results show that most proxy contracts manage storage by inheritance storage contracts. Moreover, it is rare to see a function collision issue with a different function name in practice, at least not in our dataset.



## Chapter 7 In-depth Analysis

Rank\Dataset	SC17	SC18	SC19
Top 1	79 times 0x6ab9dd83108698b9ca8d03af3c7eb91c0e54c3fc	712 times 0x0f32732e4885f0dd61b64eeef144329eb809a96e1	524 times 0xf9e266af4bca5890e2781812cc6a6e89495a79f2
Top 2	1 times 0xc48717aefbd7fbb081c48b5abdadc6a30368e75	311 times 0x072461a5e18f444b1cf2e8dde6dfb1af39197316	88 times 0x4e201a5a5534bb334a3d7df4c82cd5db3bd82f29
Top 3	0 times	299 times 0xc3b2ae46792547a96b9f84405e36d0e07edcd05c	54 times 0xb1dd690cc9af7bb1a906a9b5a94f94191cc553ce
Top 4	0 times	274 times 0x837e85498f90f9320273d2a328b5ab402b24eed6	47 times 0x4a6ce97a84178a84c1cee46a763db619d0e6e413
Top 5	0 times	169 times 0xc89327da549c6eb96c59764b13013467d17c7c79	35 times 0xf98ee39029c0f57b7d1d85be0b5579f813a58308

Table 7.1: The top 5 most pointed logic contracts (part 1).

Rank\Dataset	SC20	SC21	SC22
Top 1	1764 times 0x989a2ad9acaa8c4e50b2fc6b650d6e1809b9195b	6552 times 0x39778bc77bd7a9456655b19fd4c5d0bf2071104e	44934 times 0xf9e266af4bca5890e2781812cc6a6e89495a79f2
Top 2	1644 times 0xef004d954999eb9162aeb3989279eff2161d5095	3251 times 0x7186123dd9140555d0c2384b36f5773ddd7cde31	21411 times 0x059ffa9fd66ef594230de44f824e2bd0a51ca5ded
Top 3	992 times 0x20af9e54a3670ef6a01bca1f1ec22b1f93cbe23	2701 times 0xf9e266af4bca5890e2781812cc6a6e89495a79f2	3823 times 0xc38f942db7a1b4213d6213f70c499b59287b01f1
Top 4	864 times 0xb1dd690cc9af7bb1a906a9b5a94f94191cc553ce	1915 times 0x83b76b11257c4ece35370b6152f1946d49479e89	3317 times 0xc63730a73f5eb2b5e10486d20c10a451aa3c1c6dd
Top 5	685 times 0x2fcf5eddf53a33f9665d226f239d29eff3921bf7	1115 times 0x29b94b045a0b828d9eb99136a16d97c7f3d2600	2929 times 0x39778bc77bd7a9456655b19fd4c5d0bf2071104e

Table 7.2: The top 5 most pointed logic contracts (part 2).

In this section, we want to find the reason or meanings behind the result from the previous section.

From Table 6.2, we observe that the percentage of proxy contracts increases over time, from 16.48% in 2020 to 88.71% in 2022. Thus, we can see the usage of proxy contracts getting more widespread. Our first thought is that the increase in proxy contracts is because many new projects use proxy contracts as their services for upgradability. However, it contradicts the result in Table 6.3, which shows that most of the proxy contracts belong to minimal proxy contracts, and it is not upgradable proxy contracts.

As a result, we want to know what caused the increasing number of proxy contracts.



In the following sections, we first analyze the logic contracts and find that lots of logic contracts are wallet contracts. A wallet contract [34] is a contract that manages assets like ERC20 tokens. Usually, a proxy contract of a wallet contract is created by a smart contract named factory contract, which contains a clone function for creating proxy contracts of a wallet contract [34]. Then, based on our analysis of logic contracts, we examine their proxy contracts.

## 7.1 Logic Contracts

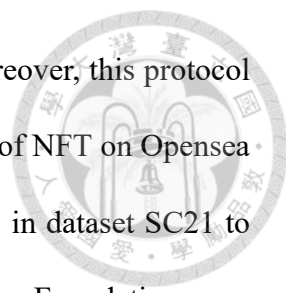
We examine all logic contract addresses and identify the top 5 logic contract addresses according to the number of associated proxy contracts. The result is shown in Table 7.1 and Table 7.2.

	Name	Address
<b>Wallet contract</b>	AuthenticatedProxy	0xf9e266af4bca5890e2781812cc6a6e89495a79f2
	CollectionContract	0xe38f942db7a1b4213d6213f70c499b59287b01f1
	Forwarder	0x059ffafdc6ef594230de44f824e2bd0a51ca5ded
	BaseWallet	0x29b94b045a0b828d9eb99136a16d97c7ff3d2600
	CloneableWallet	0x989a2ad9acaa8c4e50b2fc6b650d6e1809b9195b
	Account	0xef004d954999eb9162aeb3989279eff2161d5095
	AuthereumAccount	0x20af9e54a3670ef6a601bca1f1ec22b1f93cbe23
<b>Unknown</b>	BaseWallet	0xb1dd690cc9af7bb1a906a9b5a94f94191cc553ce
	TransactionRequestCore	0x4e201a5a5534bb334a3d7df4c82cd5db3bd82f29
	Kernel	0x4a6ce97a84178a84c1cee46a763db619d0e6e413

Table 7.3: The name of logic contracts in Table 7.1 and Table 7.2 with source code.

If a logic contract is provided with source code, we can know the name of the logic contract and its program logic.

First, we examine the contracts with source code. Table 7.3 shows the name and address of the logic contracts with the source code. The name “AuthenticatedProxy” looks like a proxy contract, but it does not fulfill our definition of proxy contracts because there is no delegatecall in its fallback function. We also find that this smart contract is from



Wyvern [36], an open-sourced decentralized exchange protocol. Moreover, this protocol is also applied by Opensea [27], an NFT trading platform. The trend of NFT on Opensea explains why the rank of the logic contract increased from the top 3 in dataset SC21 to the top 1 in dataset SC22. CollectionContract is a smart contract from Foundation open source project, which is an NFT-related project also applied by the Foundation NFT marketplace [17]. For the above two smart contracts, we also regard them as a kind of wallet contract because they both contain the logic for trading NFT assets, which acts like a wallet contract for NFT.

Forwarder is a smart contract from the open-source project Ethereum MultiSig Wallet Contract [6], which is a project for creating multi-signature wallet smart contracts. Both BaseWallet and CloneableWallet are a kind of wallet contract. Account is a smart contract from MYKEY Lab [22], which is a company for smart wallets in blockchains, but the company stopped supporting it. AuthereumAccount is a smart contract from Authereum [5] which is a company for Ethereum wallets.

TransactionRequestCore is a smart contract from Ethereum Alarm Clock [12]. Kernel is a smart contract from Aragon [2]. Except TransactionRequestCore and Kernel are not wallet contracts related, other contracts are all wallet contract related.

While the rest of the logic contracts are not provided with source code, we use an online decompiler [15] to decompile them. The table 7.4 shows the function name of the logic contract if the online decompiler recognized it, otherwise, we show the function signature instead.

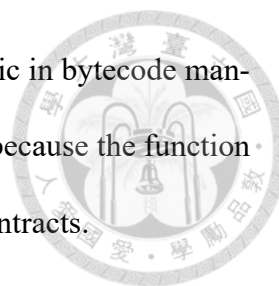
If the function names of the logic contract are as same as some functions in wallet contracts mentioned in [34], we classify them as wallet contracts. In case the logic may be



Type	address	function prototypes
Wallet contract	0x63730a73f5eb2b5e10486d20c10a451aa3c1c6dd	flushETH(), destination(), feeDestination(), init(address,address)
	0x39778bc77bd7a9456655b19fd4c5d0bf2071104e	sweeper(), sweepERC20(address), unknown(0xdfd1fb7a)
	0x7186123dd9140555d0c2384b36f5773ddd7cde310x2fcf5eddf53a33f9665d226f239d29eff3921bf7	flushERC20(address), flushETH(), destination()
	0x6ab9dd83108698b9ca8d03af3c7eb91c0e54c3fc	removeOwner(address), isOwner(address), m_numOwners(), resetSpentToday(), addOwner(address), m_required(), confirm(bytes32), setDailyLimit(uint256), execute(address,uint256,bytes), revoke(bytes32), changeRequirement(uint256), hasConfirmed(bytes32,address), kill(address), changeOwner(address,address), m_dailyLimit()
Unknown	0x83b76b11257c4ece35370b6152f1946d49479e89	unknown(0xcd6f5dcd), unknown(0xc658695c)
	0xf98ee39029c0f57b7d1d85be0b5579f813a583080x837e85498f90f9320273d2a328b5ab402b24eed6	init(address), isCosignerSet() initInsecure(address) .....
	0x0f32732e4885f0dd61b64eef144329eb809a96e1	
	0x072461a5e18f444b1cf2e8dde6dfb1af391973160xc89327da549c6eb96c59764b13013467d17c7c79	claimContractOwnership() pendingContractOwner() .....
	0xc3b2ae46792547a96b9f84405e36d0e07edcd05c	grantAccess(address) isCosignerSet() .....
	0xc48717aefbd7fbbb081c48b5abdadc6a30368e75	seriesFactory(), owner(), unknown(0x95f770fd)

Table 7.4: The function prototypes of logic contracts in Table 7.1 and Table 7.2 without source code.





different but with the same function name, we also compare their logic in bytecode manually. Eight smart contracts can not be classified as wallet contracts because the function name is unknown or does not match any known function in wallet contracts.

After our examination of the top 5 most pointed logic contracts, we find that in the most recent datasets SC20, SC21 and SC22, only one logic contract is not a wallet contract. In dataset SC17, there are 90% of logic contracts is wallet contract. While, in datasets SC18 and SC19, we do not find any specific kind of smart contract containing the most. This result answers our second research question.

<b>Types\Dataset</b>	<b>SC20</b>	<b>SC21</b>	<b>SC22</b>
Same bytecode	502 (80.45%)	1140 (81.2%)	1614 (81.2%)
Others	122 (19.55%)	264 (18.8%)	374 (18.8%)
<b>Total</b>	<b>624</b>	<b>1404</b>	<b>1988</b>

Table 7.5: The number of distinct logic contract addresses in two datasets.

We also analyze the number of distinct logic contracts. The numbers of distinct logic contract address in SC17, SC18, and SC19 are 2, 17, and 70 respectively. Only a few of them contain exactly the same bytecode in these three datasets. While for the datasets SC20, SC21, and SC22, the numbers of distinct logic contract addresses are 624, 1404, and 1988, respectively. Moreover, up to 80% of distinct logic contracts contain exactly the same bytecode in these three datasets. The result shows in Table 7.5. From this result, we can know that most logic contracts are exactly the same in the recent three years and that there are many duplicated wallet contracts.



## 7.2 Proxy Contracts

In this section, we focus on the proxy contracts themselves. As we did on logic contracts, we identify the top 5 proxy contracts in each dataset. Contracts with the same bytecode are counted together. We also analyze their bytecode or source code if provided.

The result is shown in Table 7.6 and Table 7.7. They answer our third research question.

Rank\Dataset	SC17	SC18	SC19
Top 1	79 contracts minimal proxy	712 contracts minimal proxy	524 contracts OwnableDelegateProxy
Top 2	1 contracts unknown proxy	311 contracts minimal proxy	88 contracts minimal proxy
Top 3	0 contracts	299 contracts minimal proxy	59 contracts Proxy
Top 4	0 contracts	274 contracts minimal proxy	47 contracts KernelProxy
Top 5	0 contracts	169 contracts minimal proxy	40 contracts PayingProxy

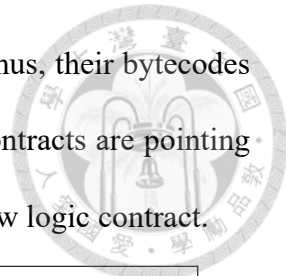
Table 7.6: The top 5 proxy contracts with the same bytecode and their contract name. (part 1)

Rank\Dataset	SC20	SC21	SC22
Top 1	1764 contracts minimal proxy	6552 contracts minimal proxy	44934 contracts OwnableDelegateProxy
Top 2	1651 contracts AccountProxy	3251 contracts minimal proxy	21411 contracts minimal proxy
Top 3	1045 contracts AuthereumProxy	2701 contracts OwnableDelegateProxy	3823 contracts minimal proxy
Top 4	866 contracts Proxy	1915 contracts minimal proxy	3317 contracts minimal proxy
Top 5	685 contracts minimal proxy	1033 contracts Proxy	2929 contracts minimal proxy

Table 7.7: The top 5 proxy contracts with the same bytecode and their contract name. (part 2)

Surprisingly, almost every cell in Table 7.6 and Table 7.7 can match the cell in the same position of Table 7.1 and Table 7.2, which is a one on one relationship. It shows that a logic contract is usually pointed by identical proxy contracts. The result supports that

the wallet contracts are usually created by a factory contract [10]. Thus, their bytecodes are the same. For the inconsistent cells, it means that other proxy contracts are pointing to the same logic contract or the proxy contract was upgraded to a new logic contract.



Proxy Contract Name	logic contract address
OwnableDelegateProxy	0xf9e266af4bca5890e2781812cc6a6e89495a79f2 (AuthenticatedProxy)
(Minimal Proxy) No Name	0x059ffa4dc6ef594230de44f824e2bd0a51ca5ded (Forwarder)
	0xe38f942db7a1b4213d6213f70c499b59287b01f1 (CollectionContract)
	0x63730a73f5eb2b5e10486d20c10a451aa3c1c6dd
	0x39778bc77bd7a9456655b19fd4c5d0bf2071104e
	0x7186123dd9140555d0c2384b36f5773ddd7cde31
	0x83b76b11257c4ece35370b6152f1946d49479e89
	0x989a2ad9acaa8c4e50b2fc6b650d6e1809b9195b
	0x2fcf5eddf53a33f9665d226f239d29eff3921bf7
	0x4e201a5a5534bb334a3d7df4c82cd5db3bd82f29 (TransactionRequestCore)
	0x0f32732e4885f0dd61b64eef144329eb809a96e1
	0x072461a5e18f444b1cf2e8dde6dfb1af39197316
	0xc3b2ae46792547a96b9f84405e36d0e07edcd05c
	0x837e85498f90f9320273d2a328b5ab402b24eed6
	0xc89327da549c6eb96c59764b13013467d17c7c79
0x6ab9dd83108698b9ca8d03af3c7eb91c0e54c3fc	
Proxy	0x29b94b045a0b828d9eb99136a16d97c7ff3d2600 (BaseWallet)
	0xb1dd690cc9af7bb1a906a9b5a94f94191cc553ce (BaseWallet)
AccountProxy	0x29b94b045a0b828d9eb99136a16d97c7ff3d2600 (Account)
AuthereumProxy	0x20af9e54a3670ef6a601bca1f1ec22b1f93cbe23 (AuthereumAccount)
KernelProxy	0x4a6ce97a84178a84c1cee46a763db619d0e6e413 (Kernel)
PayingProxy	0xf98ee39029c0f57b7d1d85be0b5579f813a5830

Table 7.8: Proxy contract names and their logic contract addresses.

We combine the proxy types in Table 7.6 and Table 7.7 with the logic contract addresses in Table 7.1 and Table 7.2 into Table 7.8. Most of the logic contracts in Table 7.8 are pointed by minimal proxy contracts. The result is reasonable because the minimal proxy contracts account for the majority of proxy contracts in Table 6.3. Minimal proxy

contracts are free to function or storage collisions, and their bytecode is short and simple. We think it may be why people widely use this proxy contract.



### 7.3 Other Findings

In our experiment, minimal proxy contracts were also found in February 2017, the time before EIP-1167 was proposed. We think it is because someone found a gas-saving method to deploy a contract but was lazy to propose an EIP.

From Table 6.3, we can observe that the number of proxy contracts following EIP-1822 or EIP-1967 is small. We think it is because the user does not need a specific storage slot to store the logic contract address but a state variable instead, which is much simpler. Though it is easy to use, one needs to worry about storage collisions if they change the order of state variables.

Dataset	avg # of blocks	avg times of upgrades	# of upgraded contracts
SC17	33529.0	2	1
SC18	0	0	0
SC19	11895666.79	1.36	14
SC20	149963.69	1.07	999
SC21	431435.68	2.04	96
SC22	187287.03	1.34	29

Table 7.9: Upgrade events.

We also investigate the upgrade events of proxy contracts. The result is shown in Table 7.9 and answers our fourth research question. Among our datasets, only 1139 proxy contracts have been upgraded. In addition, the proxy contracts in SC20 contain the most upgraded contracts, we think it is because the proxy contracts created in SC20 are still used. Thus, the proxy owners still upgrade them to support new features.

We notice that the average times of upgrades in all datasets are less than three times.

This result shows the upgraded event does not happen frequently. Since security issues of proxy contracts usually happen at an upgrade, the less frequent upgrade of a proxy contract also means the proxy contract is more resistant to storage collisions.





## Chapter 8 DISCUSSION

In this section, we first discuss the mitigation for the two main security issues. Then we discuss how to use a proxy contract based on our findings. Lastly, we discuss the limitations of our work.

### 8.1 Mitigation for Security Issues

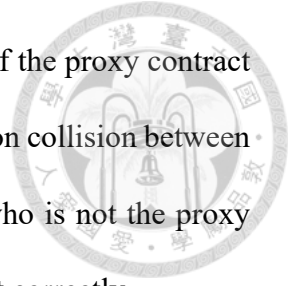
In the following two sections, we will discuss how to prevent function collisions and storage collisions from the perspectives of developers and users, respectively.

For developers, we discuss how to prevent these two types of issues when developing a proxy contract. To address the function collision problem, OpenZeppelin [25] proposes a solution called `TransparentUpgradeableProxy`. It applies a modifier<sup>1</sup> that checks if the caller of the transaction is the proxy contract owner or not on functions defined in the proxy contract. If the caller is not the proxy contract owner, it executes the fallback function instead of the current function. A developer can safely design a proxy contract without worrying about function collisions. Specifically, this kind of proxy contract only allows the owner of the proxy contract to execute functions defined in the proxy contract, but the owner of the proxy contract can not execute the fallback function of the proxy contract.

---

<sup>1</sup>A decorator applies on functions in Solidity.

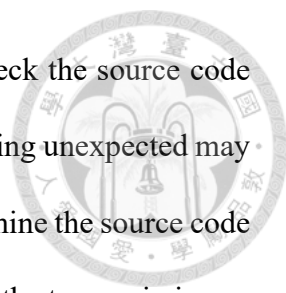
On the other hand, everyone can only execute the fallback function of the proxy contract except the owner of the proxy contract. Thus, even if there is a function collision between the proxy contract and the logic contract, a caller of a transaction who is not the proxy owner can expect to execute the function defined in the logic contract correctly.



However, some developers do not want to redesign the proxy contracts, or the proxy contracts have been deployed, so the above solution can not be applied. As a result, for those proxy contracts, we suggest that developers should run the slither-check-upgradeability tool mentioned in Section 2. The tool can check if any function collisions happen between the proxy contract and the logic contract so the developers can fix the logic contract or the proxy contract. If the proxy contracts are already deployed, developers can also use our tool by giving the proxy contract address.

Currently, the way to address storage collisions is to rely on proxy developers and proxy owners. The developers have to ensure the order of variables in the proxy contract is consistent with the logic contract. When upgrading the proxy contract, proxy owners should also ensure that the order of variables in the old logic contract is consistent with the new one. The slither-check-upgradeability tool can help developers and proxy owners check if there is any storage collision problem before deploying or upgrading a proxy contract.

Two design patterns for storage management motioned in [19], which are Unstructured Storage and Eternal Storage. Unstructured Storage uses a random storage slot to store each variable. EIP-1822, EIP-1967, and EIP-2535 apply this pattern to store some variables like a logic contract address. Eternal Storage uses another contract to store variables, making it easy for developers to maintain the order of variables.



For users who want to interact with a smart contract should check the source code of the smart contract before sending a transaction. Otherwise, something unexpected may happen, like losing Ether or other tokens. Likewise, users should examine the source code of the proxy contract before interacting with it. Moreover, because of the two main issues mentioned above, users should not only examine the proxy contract but also the current logic contract and every logic contract the proxy contract used before.

Verifying every logic contract the proxy contract used before may be challenging for users. Thus, we suggest that users can apply our tool to the proxy contract before they interact. Users only need to input the address of a proxy contract, and the tool will output if there is any security issue in the proxy contract and the source code if provided.

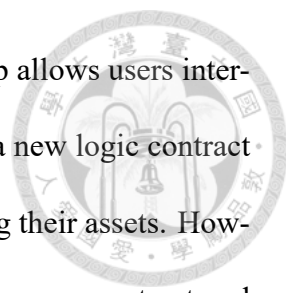
Attackers who want to leverage our tools may find some function or storage collisions, but they still have to analyze the source code to check if proxy contracts are exploitable.

## 8.2 Proxy Contract Usage

This section suggests the usage of proxy contracts from the security perspective.

As we mentioned in Section 4, the owner of the upgradable proxy contract can manipulate the storage of the proxy contract by upgrading a new logic contract. Thus, users should prevent interacting with proxy contracts not owned by themselves but holding their property, such as the ERC20Upgradeable token contract. Users should only interact with it if they trust the owner of the proxy contract. A way to improve the security of these kinds of proxy contracts is to add a time gap after the transaction of upgrading the proxy contract to a new logic contract is sent. The old logic contract is used inside the time gap,





and the new logic contract will be used after the time gap. A time gap allows users interacting with the proxy contract to examine the new logic contract. If a new logic contract is malicious, the users can react before it is effective, like withdrawing their assets. However, it is still dangerous for users if they do not keep an eye on the proxy contract and miss the time gap. Thus, we still do not recommend this kind of proxy contract usage.

Another way to use a proxy properly is to treat a proxy contract as a testing smart contract. Developers can deploy a proxy contract for an under-developing logic contract. Instead of deploying a logic contract on the testnet, an Ethereum-like environment, developers can test and interact with the logic contract on the mainnet. A testnet is an Ethereum-like environment, but its state is not the same as the Ethereum mainnet. For example, an address may be a smart contract address in the testnet but not in the mainnet. Thus, deploying a proxy contract on the mainnet for testing logic contracts can get more accurate results than on the testnet. Moreover, because of storage collisions, the proxy owner can initialize the storage without deploying a new smart contract to get new storage. Unfortunately, this method still needs to spend the gas for deploying a new logic contract.

The best usage of a proxy contract is to create one's proxy contract for personal use, such as a proxy contract for a wallet contract, so there is no need to worry about a malicious proxy contract owner. Based on our investigations, this kind of usage is the most common way to use a proxy contract. The advantage of a proxy contract is that one can save the cost of deploying a new logic contract and enable upgradeability for future needs. Moreover, if users use it as a proxy contract for wallet contracts, they can save their assets in it, and no need to transfer them after upgrading the proxy contracts.

### 8.3 Limitations



As we explained earlier, some opcodes (block opcodes) take parameters from the blockchain state. ProxyChecker uses fixed values listed in Table 5.1 as their parameters, so the emulation result may differ from the actual one if it encounters block opcodes.

ProxyChecker may fail to detect some types of proxy contracts, such as the diamond contract introduced in EIP-2535. Although the diamond contract satisfies our proxy contract definition, ProxyChecker cannot detect diamond contracts because it cannot create a calldata that can successfully execute the delegatecall opcode in the fallback function. In the design of the diamond contract, only the function registered by the diamond contract owner can execute the delegatecall in the fallback function, or the transaction will revert.

Currently, ProxyChecker leverages the slither-check-upgradeability tool for some security checks. Thus limited by this tool, ProxyChecker can only check function and storage collisions on the proxy contract with the source code in Solidity. We may support proxy contracts without source code in our future work.

The output of ProxyChecker may contain some false positives because of the implementation of slither-check-upgradeability. Nevertheless, we think it is acceptable because the harm of false negatives is much higher than false positives.

We believe ProxyChecker can complement other smart contract analysis tools.



## Chapter 9 CONCLUSION

Proxy contracts enable upgradeability, but they also bring some uncertainty for their users. Though proxy contracts are a way for smart contract developers to upgrade their vulnerable contracts, two main security issues come behind them: function collisions and storage collisions.

We introduce ProxyChecker, which is a dynamic analysis tool for smart contracts. One can use it to detect if a smart contract is a proxy contract and get a security analysis of a proxy contract by just inputting a smart contract address. However, an attacker can also use it to detect if there are any chances to attack a proxy contract.

We use ProxyChecker to compare six block ranges to show how widely proxy contracts are used. Finally, we discuss the security issues in proxy-related EIPs and give recommendations. We found that the proxy contract is a big part of all deployed smart contracts. After we categorized them, we found that the minimal proxy contracts are the most widely used. We also find that the trend of NFT creates a large number of proxy contracts. Though our ProxyChecker is an easy-to-use tool for detecting or doing a security check on proxy contracts, there still have some limitations mentioned in Section 8.3, which will be our future work.

In conclusion, if a proxy contract is upgradeable, the proxy owner can manipulate

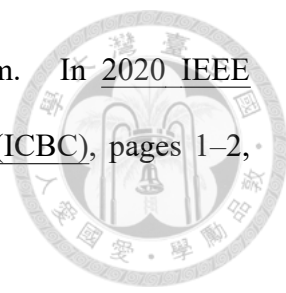
the storage value in the proxy contract to any other value. Thus, a proxy contract is not suitable for providing a smart contract service like an ERC20Upgradeable token contract. We conclude that a proxy contract is more suitable for personal use, like a proxy contract for a wallet contract.



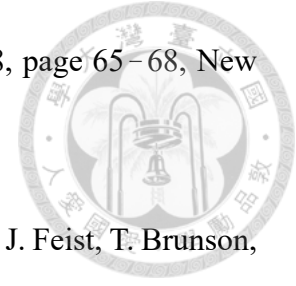


## References


- [1] Python Software Foundation. Python. <https://www.python.org/>.
- [2] Aragon Association. Aragon. <https://aragon.org/>.
- [3] Audius Inc. Audius. <https://audius.co/>.
- [4] Audius Inc. Audius governance takeover post-mortem 7/23/22. <https://blog.audius.co/article/audius-governance-takeover-post-mortem-7-23-22>.
- [5] Authereum Inc. Authereum. <https://authereum.com/>.
- [6] BitGo Inc. Ethereum multisig wallet contract. <https://github.com/BitGo/eth-multisig-v2>.
- [7] T. Chen, Z. Li, X. Luo, X. Wang, T. Wang, Z. He, K. Fang, Y. Zhang, H. Zhu, H. Li, Y. Cheng, and X.-s. Zhang. Sigrec: Automatic recovery of function signatures in smart contracts. IEEE Transactions on Software Engineering, pages 1–1, 2021.
- [8] Consensys Inc. Mythril. <https://github.com/ConsenSys/mythril>.
- [9] David, S. Understanding the dao attack. <https://www.coindesk.com/learn/2016/06/25/understanding-the-dao-attack/>.

- 
- [10] M. di Angelo and G. Slazer. Wallet contracts on ethereum. In 2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), pages 1–2, 2020.
- [11] EOSIO Inc. Eos. <https://eos.io/>.
- [12] Ethereum Alarm Clock. Ethereum alarm clock. <https://www.ethereum-alarm-clock.com/>.
- [13] Ethereum community. Opcodes for the evm. <https://ethereum.org/en/developers/docs/evm/opcodes/>.
- [14] EtherScan Team. Etherscan. <https://etherscan.io/>.
- [15] ethervm@gmail.com. Online solidity decompiler. <https://ethervm.io/decompile>.
- [16] J. Feist, G. Greico, and A. Groce. Slither: A static analysis framework for smart contracts. In Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB '19, page 8–15. IEEE Press, 2019.
- [17] foundation Inc. Foundation. <https://foundation.app/>.
- [18] Infura Inc. Infura. <https://infura.io/>.
- [19] P. Klinger, L. Nguyen, and F. Bodendorf. Upgradeability concept for collaborative blockchain-based business process execution framework. In International Conference on Blockchain, pages 127–141. Springer, 2020.
- [20] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe. Reguard: Finding reentrancy bugs in smart contracts. In Proceedings of the 40th International Conference

on Software Engineering: Companion Proceedings, ICSE '18, page 65–68, New York, NY, USA, 2018. Association for Computing Machinery.



- [21] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 1186–1189, 2019.
- [22] MYKEY Lab. Mykey lab. <https://mykey.org/>.
- [23] Neo Inc. Neo. <https://neo.org/>.
- [24] R. Norvill, B. B. F. Pontiveros, R. State, and A. Cullen. Visual emulation for ethereum’s virtual machine. In NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium, pages 1–4, 2018.
- [25] OpenZeppelin Inc. Proxy upgrade pattern. <https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies>.
- [26] OpenZeppelin Inc. Safemath. <https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/math/SafeMath.sol>.
- [27] Ozone Networks Inc. Opensea. <https://opensea.io/>.
- [28] P. Ventuzelo. Octopus. <https://github.com/pventuzelo/octopus>.
- [29] QuickNode Inc. Quicknode. <https://www.quicknode.com/>.
- [30] Rekt DAO. Rekt. <https://rekt.news/>.

- 
- [31] Solidity community. Solidity official document. <https://docs.soliditylang.org/>.
- [32] The go-ethereum Authors. Go ethereum. <https://geth.ethereum.org/>.
- [33] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In 2021 IEEE European Symposium on Security and Privacy (EuroS&P), pages 103–119, 2021.
- [34] Trail of Bits Inc. rattle. <https://github.com/crytic/rattle>.
- [35] Vyper community. Vyper official document. <https://vyper.readthedocs.io/en/stable/>.
- [36] Wyvern Protocol team. Wyvern protocol. <https://wyvernprotocol.com/>.
- [37] Yiedld App Inc. Yield. <https://www.yield.app/>.
- [38] G. Zheng, L. Gao, L. Huang, and J. Guan. Upgradable contract. In Ethereum Smart Contract Development in Solidity, pages 197–213. Springer, 2021.