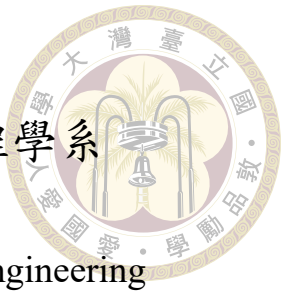


國立臺灣大學電機資訊學院資訊工程學系
碩士論文
Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Taiwan University
Master's Thesis

Etherspect: 用於探索所有以太坊代理合約的自動化可擴展智慧合約分析平臺

Etherspect: An Automated and Scalable Smart Contract Analysis Platform for Discovering All Proxy Contracts in Ethereum

陳正康
Zac Chengkang Chen

指導教授: 蕭旭君 博士
Advisor: Hsu-Chun Hsiao, Ph.D.

中華民國 113 年 8 月
August, 2024

doi:10.6342/NTU202401985

國立臺灣大學碩士學位論文
口試委員會審定書

MASTER'S THESIS ACCEPTANCE CERTIFICATE
NATIONAL TAIWAN UNIVERSITY

Etherspect：用於探索所有以太坊代理合約的自動化可擴展智慧合約分析平臺

Etherspect: An Automated and Scalable Smart Contract Analysis Platform for Discovering All Proxy Contracts in Ethereum

本論文係陳正康君（學號 R10922187）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 113 年 6 月 25 日承下列考試委員審查通過及口試及格，特此證明。

The undersigned, appointed by the Department of Computer Science and Information Engineering on 25 June 2024 have examined a Master's thesis entitled above presented by ZAC CHENGKANG CHEN (student ID: R10922187) candidate and hereby certify that it is worthy of acceptance.

口試委員 Oral examination committee:

蕭旭君	黎士瑋	
（指導教授 Advisor）		

系主任/所長 Director: 陳祝嵩

國立臺灣大學碩士學位論文
口試委員會審定書

MASTER'S THESIS ACCEPTANCE CERTIFICATE
NATIONAL TAIWAN UNIVERSITY

Etherspect：用於探索所有以太坊代理合約的自動化可擴展智慧合約分析平臺

Etherspect: An Automated and Scalable Smart Contract Analysis Platform for Discovering All Proxy Contracts in Ethereum

本論文係陳正康君（學號 R10922187）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 113 年 6 月 25 日承下列考試委員審查通過及口試及格，特此證明。

The undersigned, appointed by the Department of Computer Science and Information Engineering on 25 June 2024 have examined a Master's thesis entitled above presented by ZAC CHENGKANG CHEN (student ID: R10922187) candidate and hereby certify that it is worthy of acceptance.

口試委員 Oral examination committee:

蕭旭君		
（指導教授 Advisor）		
	Dr. Muoi Tran	

系主任/所長 Director: 陳祝嵩

誌謝

首先感謝蕭旭君老師的指導。老師的指導風格非常的彈性自由，讓我有空間探索並找到有興趣的題目，並在構思與實作中培養研究的能力。感謝老師對我的題目的支持和建議，讓我逐漸理清原本雜亂不太成熟的想法，讓它們變成一個相對完整有價值的研究。最後在口試準備期間也謝謝忙碌的老師花費不少時間給我許多有用的意見，讓我不太擅長的口頭發表更順暢。

同時感謝一起共事的 Muoi 在設計實驗時給了很多寶貴建議，協助我把方法設計得完整有條理。祝福他在瑞典的學術生涯順遂。

感謝實驗室的同學們在我最忙碌的口試前一個月陪我度過，包括一起奮鬥口試和論文的義閔、幫忙測試平台和練習口試的子民、提供教授級專業意見的完美超人承諺、啟發我的論文題目的奇傘、被我拖去 FLOLAC 的哲睿和柚昇、優秀的學者楷翔、帶我進入 DeFi 領域的家謙、舉辦各種實驗室活動的俐霏和寶霈、還有沒列到的其他人。

最後感謝家人的支持和女友雅涵的陪伴和打理生活。他們是我碩士生涯背後的支柱。

在 NSLab 的 2 年半很自由、多元、有趣，也學到很多東西。祝實驗室能蓬勃發展，有更多優秀的人加入，產出更多有趣的研究。祝老師和同學們平安、順利。

doi:10.6342/NTU202401985

doi:10.6342/NTU202401985

摘要

以太坊智慧合約的代理設計模式將合約狀態和程式碼邏輯分別解耦至代理合約和邏輯合約中，為之引入了程式碼更新機制、功能重用和程式碼模組化。然而，這種靈活的設計模式也引入了新的安全風險，即函數名稱衝突和儲存位置衝突。面對現實世界中已造成數百萬美元損失的攻擊，業界和學術界對這些問題進行了研究並尋求解決方案。然而，現有研究或涵蓋不足，或依賴原始碼與區塊鏈歷史，未能全面了解以太坊上的代理合約和衝突問題。

為填補此研究缺口，我們提出了涵蓋以太坊「所有」代理合約進行全面研究的方法。我們發現現今缺乏易用且高效的解決方案來評估大型的智慧合約資料集。現有的方法或未能適當處理資料集以進行大規模分析，或無法在分散式計算環境中擴展，或未能妥善管理複雜的分析程序。

為提高效率，我們開發了 ETHERSPECT ，一個自動化且可擴展的智能合約分析平台，不僅用於分析代理合約，還能執行一般的智慧合約分析工具。ETHERSPECT 在效率、擴展性和自動化方面表現優越。藉由 ETHERSPECT 的預處理資料集和可擴展的依賴關係感知分析工具排程器，對所有以太坊代理合約進行的複雜大規模分析預計將從 2,228 天縮短至 25 天。最終，ETHERSPECT 分析了 3700 萬個乙太坊智慧合約，發現了 2000 萬個代理合約、150 萬個函數名稱衝突和 2.5 萬個儲存位置衝突。

關鍵字：智慧合約、智能合約、代理合約、區塊鏈安全、軟體測試、分散式計算

doi:10.6342/NTU202401985

doi:10.6342/NTU202401985

Abstract

The evolving proxy design patterns bring upgradability, functionality cloning, and modularity to Ethereum smart contracts by decoupling contract state and code logic into a proxy contract and a logic contract, respectively. This flexible pattern introduces new security risks, namely *function collisions* and *storage collisions*. In the presence of real-world attacks with millions of dollars worth of loss, the industry and the academy have studied the issues and searched for solutions. However, previous research either lacks enough coverage or relies on source code or past transactions, failing to understand all proxy contracts and collision issues comprehensively.

To address the gap, we proposed a complete study of all proxy contracts in Ethereum. Simultaneously, we found the lack of a user-friendly and efficient solution to evaluate large smart contract datasets. Current solutions either do not properly tailor datasets for large-scale analysis, do not scale out in a distributed computing environment, or do not manage complex analysis procedures appropriately.

For efficiency, we built EtherSpect, an automated and scalable smart contract anal-

doi:10.6342/NTU202401985

ysis platform, for not only analyzing proxy contracts but also executing general smart contract analyzers. ETHERSPECT is shown to excel in efficiency, extensibility, and automation. With ETHERSPECT's preprocessed dataset and scalable dependency-aware analyzer scheduler, a complicated large-scale analysis to discover and analyze all Ethereum proxy contracts is estimated to reduce from 2,228 days to 25 days, finally analyzing 37M Ethereum smart contracts, identifying 20M proxy contracts, 1.5M function collisions and 25k storage collisions.

Keywords: smart contracts, proxy contracts, blockchain security, software testing, distributed computing

doi:10.6342/NTU202401985

Contents

doi:10.6342/NTU202401985

doi:10.6342/NTU202401985

doi:10.6342/NTU202401985

doi:10.6342/NTU202401985

List of Figures

doi:10.6342/NTU202401985

doi:10.6342/NTU202401985

List of Tables

doi:10.6342/NTU202401985

doi:10.6342/NTU202401985

Chapter 1 Introduction

Ethereum is a well-established blockchain system that facilitates various decentralized applications, including decentralized finance (DeFi), non-fungible tokens (NFTs), and e-voting. Applications on Ethereum operate based on smart contracts, immutable code that are deployed by developers and executed by Ethereum nodes running Ethereum Virtual Machine (EVM). By the fundamental design of Ethereum, smart contract codes are immutable once deployed to ensure integrity and reliability. However, this immutability has led to great challenges for developers to fix bugs and introduce new features to existing applications. As a result, developers have adopted several design patterns that enable upgradability [1].

Among these upgradable design patterns, the proxy pattern is the most popular [1]. The Ethereum community has also established several standards for proxy patterns as Ethereum Request for Comments (ERC) ([2, 3, 4, 5, 6]). In the proxy pattern, smart contracts are separated into 2 contracts: a *proxy contract* that stores the states, and a *logic contract* that keeps the implementation logic. These 2 contracts interact though *delegate call*, a special operation in Ethereum that is used to delegate the execution to the code of the logic contract while using the context and states of the proxy contract. Upgrades are simply done by replacing the logic contract. The flexibility of the proxy pattern provides not only upgradability but also code reuse and modularity, making it one of the most popular design patterns. According to our statistics, in 2023, over 90% of newly deployed

doi:10.6342/NTU202401985

smart contracts use proxy patterns.

The proxy patterns also introduce new security issues. Among them, function collisions and storage collisions are the most significant ones. Specifically, as the proxy pattern requires binding 2 different smart contracts at runtime, one being the proxy contract and the other being the logic contract, function signatures or storage layout may collide in this architecture if developers are not careful. These collisions open vulnerabilities to fraud and attackers. In a real-world attack on Audius [7, 8], storage collisions have led to 1.1 million worth of funds.

With such severe security impacts, proxy contracts and their potential collision problems have been the focus of recent research. However, most of them have limited coverage, or rely on historical information. Among them, Slither [9] simply searches for relevant keywords such as “proxy” and “upgradable” in the smart contract source code to identify proxy contracts. The limitation of Slither is that it does not discover logic contracts as it does not look into proxy contracts’ state. USCHunt [10] extended Slither by adding cross-contract dataflow analysis and found out 1% of the smart contracts in Ethereum are proxy contracts. Then it discovers logic contracts by looking for previous delegate calls from the proxy contracts in historical transactions. Both Slither and USCHunt have limited coverage to only smart contracts with source code available, which account for only less than 10% until May 8th, 2024 (Figure 1.1). Crush [11], on the other hand, performs bytecode and transaction analysis and does not rely on source code. It finds proxy contracts by graph analysis on past delegate call interactions, similar to the approach of USCHunt. Then it does type inference on storage slots after decompilation on the bytecode to find storage collisions and use symbolic execution to filter out non-exploitable ones. Despite having a wider coverage, Crush still misses proxy contracts that happen to

doi:10.6342/NTU202401985

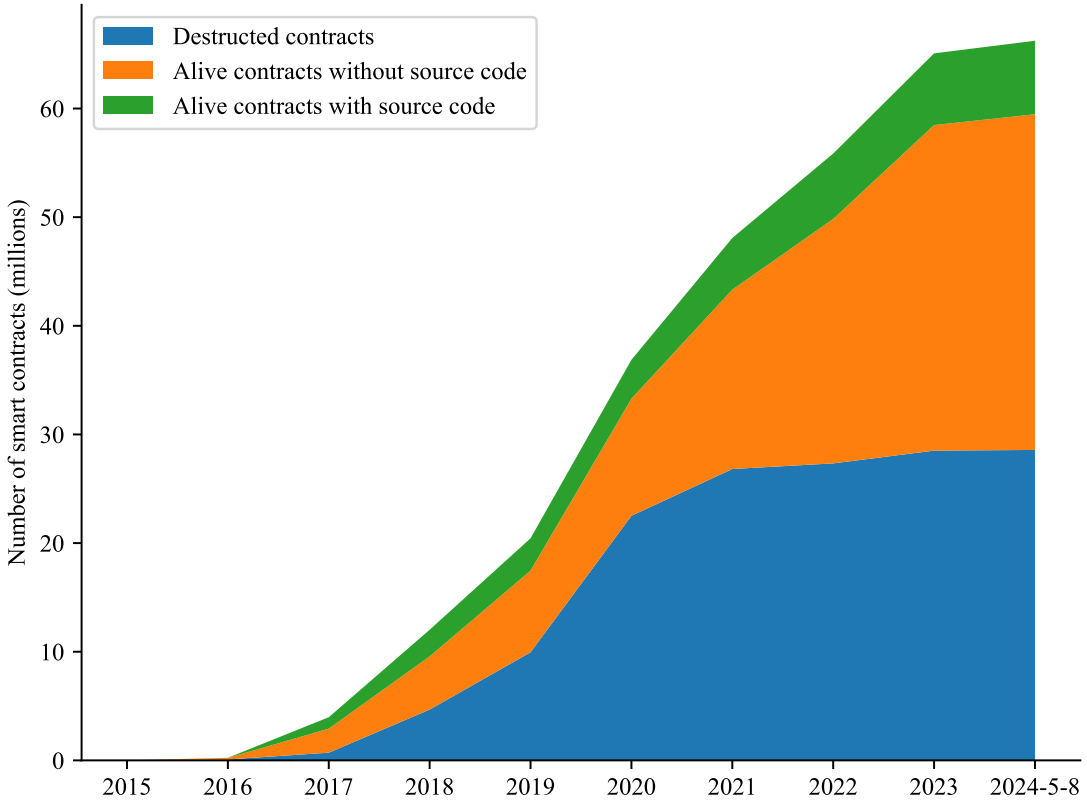


Figure 1.1: Accumulated number of destructed and alive Ethereum smart contracts until May 8th, 2024.

have no delegate call interactions in history, which happens especially to newly deployed proxy contracts. ProxyChecker [12] is another bytecode-based analyzer that detects proxy contracts and discovers their logic contracts by dynamic analysis on bytecode and contract states, requiring no historical information. However, due to the time limit, their evaluation was limited to 0.5M sampled contracts. The limited coverage of these previous work suggests a need for a comprehensive study of *all* proxy contracts in Ethereum.

To conduct a large-scale study on all proxy contracts in Ethereum, we use ProxyChecker [12] to discover all proxy contracts throughout 37 million alive smart contracts

doi:10.6342/NTU202401985

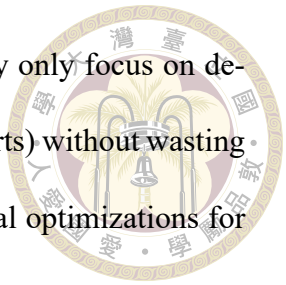
up to May 8th 2024. ProxyChecker also identifies common proxy types and standards, such as minimal proxy contracts [3]. After finding out all proxy contracts and logic contracts, we then use Slither [9] and Crush [11] to detect collisions.

In general, a large-scale study on Ethereum takes much effort. First, a complete smart contract dataset up to the latest date is needed (until May 8th, 2024, there have been 66 million smart contracts deployed). The dataset should contain smart contract data required for analysis: bytecode, source code, transactions, or other relevant data depending on the method used in the study. In Ethereum, however, all data are kept in the unit of transactions wrapped in blocks on archive nodes. A transaction parser is needed to extract meaningful data from an Ethereum archive node, such as the address and bytecode of a deployed contract. Besides, smart contract source codes are not kept on Ethereum nodes but instead on Etherscan [13], a third-party blockchain explorer service. Different data sources complicate the preparation of a complete dataset.

Second, researchers often need to develop some smart contract analyzers that analyze data from the dataset and produce meaningful reports for their study. Efficiently scheduling analyzer instances to fully utilize all available computing resources is crucial to reducing the time required to complete a large-scale analysis. To picture the time needed to complete a large-scale analysis, Angelo et al. [14] spent up to 30 CPU years to complete a large-scale evaluation of 12 smart contract analyzers on a complete dataset up to January 2022. That emphasizes the need for accelerating the evaluation process through various techniques. One of the common techniques is pruning the dataset to remove contracts with identical bytecode [14]. For Ethereum in 2024, this reduces the dataset size greatly by 97%.

For the reasons above, we find it worth studying to improve such a large-scale anal-

doi:10.6342/NTU202401985



ysis process on Ethereum. We believe that researchers should ideally only focus on designing core methodology (i.e. develop analyzers and inspect the reports) without wasting too much time on the preparation of a general dataset or doing general optimizations for speeding up.

Unfortunately, available smart contract datasets and analysis frameworks are not perfectly tailored for large-scale analysis. They either require too much manual effort on preprocessing or do not handle analyzer dependencies and duplicated contracts well enough, compromising analysis efficiency. Smart Contract Sanctuary [15] claims to maintain a GitHub repository of all smart contracts with verified source code on Etherscan. It used to be a popular dataset in the academy [10], but at the time of writing, the last update to this repository was 1 year ago, missing almost half of the available smart contracts as of 2024. This reveals a significant drawback of manually maintaining a smart contract dataset, as Ethereum has an average of 20 thousand smart contracts deployed and more than 200 having their source code verified every day [16]. The Ethereum public dataset on Google BigQuery [17, 18, 19], on the other hand, is an automatically up-to-date dataset by extracting deployed contract address and bytecode from an Ethereum archive node every day. However, it misses source code, which requires additional efforts to download source code from another source such as Etherscan [13]. Besides, it does not recognize duplicated bytecode at all, making large-scale analysis inefficient.

SmartBugs [20, 21] is a smart contract analysis framework with 20 integrated analyzers and prebuilt datasets, with a goal most closely related to ours. It provides an analyzer scheduler that feeds contract data to analyzers and manages and executes analyzer instances. The datasets provided in SmartBugs are preprocessed for large-scale analysis, including bytecode deduplication and bytecode skeletonize (trimming nonfunc-

doi:10.6342/NTU202401985

tional portions from the bytecode) [22, 14]. These preprocessing steps drastically reduced the dataset size to only 0.5% of the original size, thereby saving a considerable amount of time. However, in addition to suffering from manual dataset updating efforts similar to Smart Contract Sanctuary, SmartBugs does not handle the dependencies of analyzers. In particular, when the entire analysis process involves several steps, these steps may have dependencies. Each analysis step may depend on contract states (i.e. storage and balance), or may only depend on contract bytecode. In the latter case, we can use contract bytecode from the deduplicated dataset to save the total number of contracts that need to be executed. Analyzer dependencies introduce complexity in managing inputs and scheduling analyzer instances. In our study, the analyzers are ProxyChecker, Slither, and Crush. ProxyChecker depends on contract states and thus inputs *contracts*, while Slither and Crush depend on the results of ProxyChecher, and inputs *contract bytecode*. In order to correctly orchestrate analyzer instances, a lookup table that correlates a contract to its distinct bytecode needs to be added to the analyzer scheduler during the analysis process.

In the presence of these gaps, we present EtherSpect, an automated and scalable smart contract analysis platform. EtherSpect is a complete solution that aims to make large-scale studies on Ethereum more efficient and automated. Its main features include:

- Automatically up-to-date smart contract dataset with contract address, bytecode, source code, and related transactions. There is also a subset containing distinct contract bytecode, indexed by bytecode hash.

- A scalable, dependency-aware analyzer scheduler that schedules smart contract data and analyzer instances onto processors in a cluster. The scheduler can optionally map contracts to their key contracts depending on the type of analyzer.

- An interface to integrate command-line interface (CLI)–based smart contract ana-

doi:10.6342/NTU202401985

lyzers.

- Reliable execution with error handling and execution timeout.
- An automated architecture that reduces manual intervention during the evaluation.

We evaluated EtherSpect by performance, scalability, and extensibility, and showed its capability to efficiently conduct a large-scale study on Ethereum with minimal human intervention. We also used EtherSpect to study all proxy contracts in Ethereum. As a result, we found 20M proxy contracts and their logic contracts, along with 1.5M function collisions and 25k storage collisions. Our comprehensive study successfully finished in 25 days, far more efficient than the estimated 2,228 days of a naive approach.

In summary, our contributions include:

- A comprehensive study on all proxy contracts in Ethereum, with a broader coverage than previous work, found 20M proxy-logic contract pairs, 1.5M function collisions, and 25k storage collisions. The total analysis is completed within only 25 days.
- Building an efficient framework that makes large-scale smart contract analysis more automated and efficient.

doi:10.6342/NTU202401985

doi:10.6342/NTU202401985

Chapter 2 Background

This section provides the necessary background, including smart contracts, proxy patterns of smart contracts, and their collision issues.

2.1 Ethereum and Smart Contracts

Smart contracts, first introduced by Ethereum blockchain [23], are immutable codes that are deployed and executed on the blockchain to maintain a global state on the blockchain. A smart contract is first written in high-level programming languages, the most popular ones being Solidity [24] and Vyper [25], compiled into bytecode, then deployed onto the Ethereum blockchain through transactions issued by user accounts.

Once a smart contract is deployed with a fixed bytecode, it is given a contract address, a balance in Ether (the currency in Ethereum), and a storage that keeps mutable data. Users can interact with a deployed contract by issuing transactions, which contain call data that describes a 4-byte function signature (the hash of function name and parameter types) of the target contract and function arguments. A transaction triggers some function stored in the contract's bytecode, which can manipulate storage, send Ethers, or interact with other smart contracts through external calls. Every node in the Ethereum blockchain runs an Ethereum Virtual Machine (EVM) to execute transactions. When a transaction modifies the global state (contracts' balance or storage), it is updated accordingly and synchronized

doi:10.6342/NTU202401985

across the blockchain using consensus protocols. The state change persists as the issued transactions are wrapped in a new block, which is assigned an incremental block number and appended to the blockchain.

In a low-level aspect, in the execution of a transaction, instructions that make up the bytecode are executed. Execution of each instruction consumes some amount of gas, measured in Ether, as the user's cost of issuing a transaction.

2.2 Proxy Patterns

Due to the immutability of blockchain, deployed smart contract code cannot be changed. This introduces a big challenge for upgrades on the smart contracts, fixing bugs, and providing new features. Though smart contracts can self-destruct, their states are lost and need to be migrated to another contract in advance. There are several design patterns to provide upgradability for smart contracts, among which the proxy patterns are the most popular [1]. Proxy patterns not only offer upgradability [2] but also functionality cloning [3] and modularity [6], making the proxy patterns a popular design pattern.

The proxy pattern splits a smart contract's states and code logic into a proxy contract and a logic contract, respectively. The proxy contract usually implements minimum necessary functionality such as upgrade functions, and delegates any other function call to the logic contract. A fallback function in the proxy contract is used to achieve the behavior. When no function matches the function signature required by the caller, the fallback function is always executed, where the proxy contract forwards the call to the logic contract via a *delegate call* (i.e. there is a DELEGATECALL instruction in the bytecode), a special type of external call that executes the target contract under the context (storage and balance) of the caller contract. By separating states and logic into 2 different contracts, up-

doi:10.6342/NTU202401985

grades can be achieved by simply changing the logic contract that a proxy contract points to (usually a storage variable stores the logic contract address).

Using proxy patterns, functionality cloning and modularity is simple because of proxy patterns' flexible nature. The former is achieved by *minimal proxy contracts*, which was later standardized by ERC-1167 [3]. Without minimal proxy contracts, reusing functionality from an existing smart contract typically involves *code cloning* [26], which is commonly accomplished by copying and pasting source code or generating bytecode via the CREATE instruction. This practice leads to numerous bytecode duplicates on Ethereum (Section 3.1.3), consuming a significant portion of the precious blockchain storage. The latter, on the other hand, can be achieved by multi-faucet proxy patterns, the most popular of which is the diamond proxy [6]. Modularity enables flexible upgrades and reduces development and maintenance costs from a software engineering perspective.

2.3 Function Collisions and Storage Collisions

As with any flexible software design, proxy patterns are susceptible to vulnerabilities due to careless design. Function collisions and storage collisions are the most significant ones.

Function collisions happen when some function signatures of the logic contract collide with that of the proxy contracts [27]. The behavior of a fallback function causes the proxy contract not to delegate a function call to the logic contract as expected but instead executes the matching function in the proxy. When the colliding function has different names, this collision is obscure to users. Therefore, a honeypot contract [28] can exploit function collision and users' unawareness to deceive them into executing a malicious function in the proxy contract by enticing them to call an appealing function in the logic

doi:10.6342/NTU202401985

contract, as long as these 2 functions have the same 4-byte function signature. Generating such function signature collisions (e.g. `collate_propagate_storage(bytes16)` and `burn(uint256)` and the same signature 0x42966c68) is effortless on modern computers (within seconds). To avoid function collisions, proxy contracts should be implemented with minimal functions and keep only necessary functions such as upgrade functions. Following that, transparent proxy [29] and universal upgradable proxy [4] are good proxy contract implementations.

Storage collisions, on the other hand, happen when the storage layout assumed by the proxy contract and the logic contract, or by different version logic contracts, are different, as shown in Figure 2.1. As in a proxy pattern, the proxy contract and all the logic contracts share the same storage of the proxy contract, this mismatch leads to unexpected behavior and vulnerabilities. For example, in the attack on Audius protocol [7], an uncareful developer makes the logic contract use the storage slot 0 as `bool initialized`, while the proxy contract also uses the storage slot 0 but as `address proxyAdmin`. Then due to the last byte of `address proxyAdmin` being 0, `bool initialized` is deemed false by the logic contract, where any user can trigger dangerous re-initialization. Similarly, storage collision can occur on 2 versions of the logic contract after an upgrade. To avoid storage collisions, it is recommended to enforce proxy contracts to use only specific storage slots that never collide with the logic contracts [5], and maintain existing state variable orders when introducing new state variables in a new logic contract.

doi:10.6342/NTU202401985

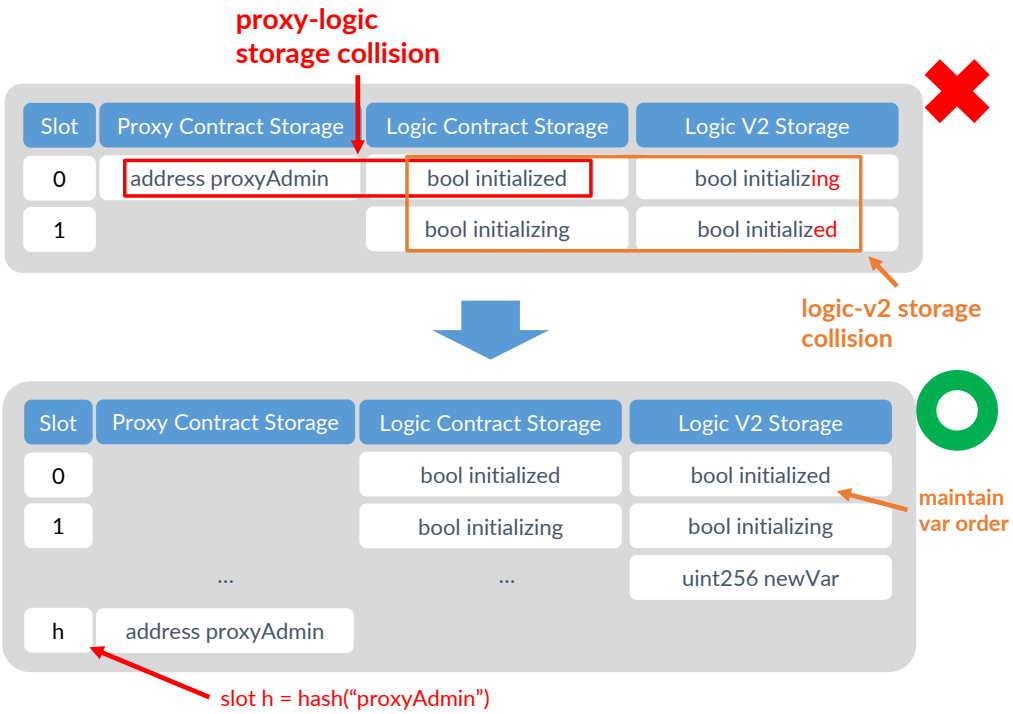


Figure 2.1: Storage collision.

doi:10.6342/NTU202401985

doi:10.6342/NTU202401985

Chapter 3 Design

This section describes the design of ETHERSPECT. In Section 3.1, we briefly describe the design goals of ETHERSPECT. ETHERSPECT is designed in a data pipeline architecture that regularly ingests data from different data sources and processes them in several components. Components for preparing a smart contract dataset include contract collector (Section 3.2) and source code retriever (Section 3.3). Components for executing smart contract analyzers include analyzer scheduler (Section 3.4) and analyzer executor (Section 3.5). There is also a built-in analyzer proxy contract detection unit (Section 3.6) to generate proxy contract information. An overview of the ETHERSPECT's architecture is presented in Figure 3.1.

3.1 Design Goals

This section elaborates on the design goals of ETHERSPECT.

3.1.1 Automated and Online Evaluation

Due to blockchain's immutable nature, it is extensively used to manage valuable assets, leading to high-security standards for smart contract applications. Consequently, significant efforts by both academia and industry have focused on evaluating and ensuring the security of these contracts, leading to the development of numerous security analysis tools [14, 30, 31]. However, the lack of readily available datasets and automated tools

doi:10.6342/NTU202401985

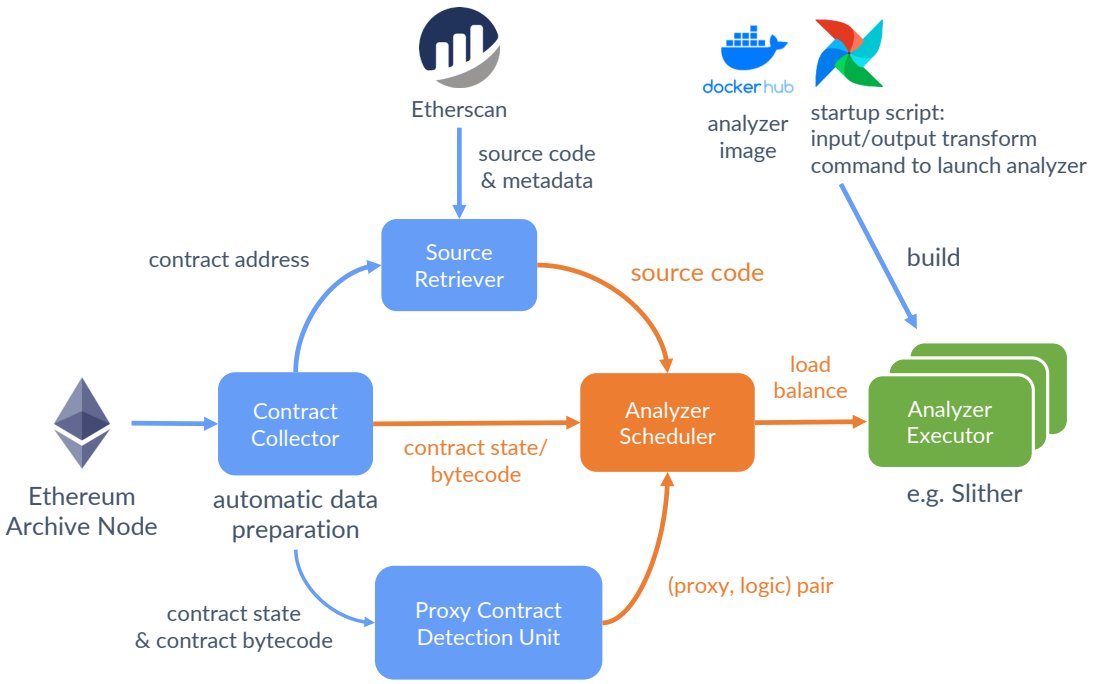


Figure 3.1: Overview of ETHERSPECT

results in considerable manual effort for data collection and processing. Additionally, real-time analysis is often required to deliver timely security alerts to users as soon as the smart contracts are deployed or upgraded. ETHERSPECT targets an automated solution for these procedures.

3.1.2 A Ready, Single Source of Data for Analysis

Most smart contract analyzers utilize bytecode, source code, transaction data, or a combination thereof for their analysis. On-chain storage is occasionally considered as

doi:10.6342/NTU202401985

well. These data come from an Ethereum archive node, or centralized online services such as Etherscan [13]. ETHERSPECT should automatically retrieve and structure these data from their sources, facilitating immediate analysis.

3.1.3 Separating *Contract Code* and *Contract* into Distinct Logical Entities

Our observations reveal significant duplication of bytecode on Ethereum, with unique bytecode representing only 2% of smart contracts deployed throughout history. This prevalence of duplicate bytecode is primarily due to code cloning, which allows smart contracts to reuse functionality while maintaining separate storage [26].

For large-scale smart contract analysis, certain analyses rely solely on bytecode or source code, rendering contract state irrelevant. Avoiding redundant analysis of identical bytecode or source code can substantially reduce processing time.

To address this, we distinguish between the concepts of *contract code* and *contract*. By hashing the bytecode of each contract, we can isolate analysis to contracts with unique bytecode hashes.

3.1.4 Integration with Smart Contract Analyzers

Most smart contract analyzers provide a command-line interface (CLI) that uses the contract address as an argument and requires specific dependencies for execution. To ensure isolated execution, ETHERSPECT employs container technology. To integrate with ETHERSPECT, each analyzer requires an "adapter" to connect the CLI with ETHERSPECT's workflow. This adapter must read inputs from the database, execute the analyzer with the appropriate arguments in a container, and write the results back to the database.

doi:10.6342/NTU202401985

3.1.5 A Scalable Analysis Platform

ETHERSPECT is designed with scalability in mind. It allows for the partitioning of analyzer instances and data to support distributed computing. Hence, the analyzer scheduler should be able to deploy instances across different machines. Consequently, ETHERSPECT relies on a network-connected database for synchronization rather than inter-process communication methods provided by operating systems.

3.2 Contract Collector

The contract collector is the first stage of the pipeline that directly ingests raw transaction data from an Ethereum archive node and extracts meaningful data about smart contracts. As ETHERSPECT keeps running on-chain, the contract collector keeps parsing the latest blocks that have not yet been processed to look for contract deployment and destruction events. At the same time, a set of distinct contract bytecode is maintained.

Usually, during analysis, a reference block number is fixed by the researchers to ensure result consistency. Though the contract deployment and destruction occur continuously, given a reference block number, the contract collector is able to generate a fixed contract dataset that includes all contracts that are still alive (not yet destructed) at the reference block for conducting the large-scale analysis.

How the contract collector works is shown in Figure 3.3. In summary, 3 lists are maintained and updated regularly: ① the deployed contract list, ② the destructed contract list and ③ the (distinct) contract code list; and additionally, a temporary list is generated on demand: ④ the alive contract list. ① and ② are indexed by contract address, while ③ is indexed by bytecode hash (Figure 3.2). ④ is the difference between ① and ② with the deployed time before the given reference block number.

doi:10.6342/NTU202401985

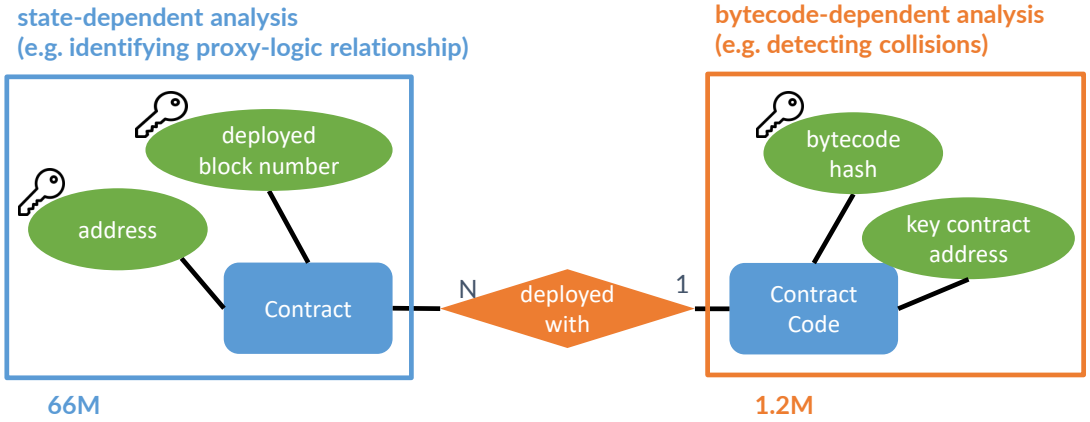


Figure 3.2: *Contract* and *contract code* relationship in a entity-relationship model. Contracts are indexed by contract address and deployed block number, whereas contract codes are indexed by bytecode hash. The key contract address of a contract code entity references one instance of the contract entity that has the code. Analysis that depends on contract state should use the *contract* dataset, while analysis that only depends on contract code can use the *contract code* dataset.

The mapping from the deployed contract list to the contract code list is an N-1 relationship (Figure 3.2). As the average N is large (almost 2 orders of magnitude in Ethereum), this becomes the key to saving large-scale analysis time. In order not to introduce modifications to analyzers, *contract code* is mapped to its key contract before executing the analyzer, and thus analyzers still see a normal contract as input, as shown in Figure 3.4.

doi:10.6342/NTU202401985

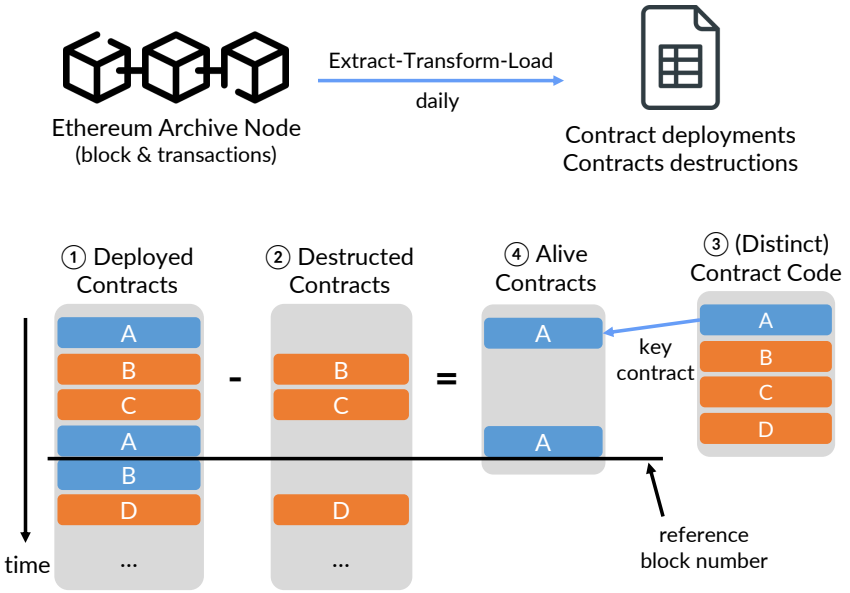


Figure 3.3: Contract collector with 4 lists: ① the deployed contract list, ② the destructed contract list and ③ the contract code list, and ④ the alive contract list. A, B, C, and D each represent a distinct bytecode. The key contract of a contract code entity refers to the first instance of the contract entity that was deployed with the bytecode. All identical bytecodes deployed later are non-key contracts of the contract code. A reference block number is used to fix the set of alive contracts that is to be used in the upcoming analysis.

3.3 Source Code Retriever

The source code retriever is a simple unit that tries to download the source code of each distinct bytecode in the contract code list from Etherscan using its API [32]. When a contract bytecode has been verified on Etherscan, it responds to the source code along with metadata and compiler configurations that are required to reproduce compilation.

doi:10.6342/NTU202401985

3.4 Analyzer Scheduler

The analyzer scheduler schedules and executes task units that consist of a list of contracts and an analyzer instance, and takes care of preparing input and retrieving output for the analyzer executor. A novel design in ETHERSPECT's analyzer scheduler is that it manages the *dependencies* among analyzers. Executing each analyzer in the correct order in ETHERSPECT is not that simple as ETHERSPECT has 2 types of input: contract and contract bytecode (Section 3.2). When an analyzer reads contract states such as storage or balance, the analyzer scheduler puts contracts from the alive contract list (Section 3.2) as input; however, when an analyzer only reads the bytecode of the input contracts, the analyzer scheduler will put contract bytecode from the contract bytecode list as input, reducing the total number of contracts that need to be run. For example, the proxy contract detection unit (Section 3.6) reads contract states and produces proxy and logic contracts. Then, they need to be mapped to correlated contract bytecode before passing to an analyzer that does not depend on contract states (e.g. a collision detector like Slither). ETHERSPECT's analyzer scheduler introduces dependency-aware scheduling that handles that mapping operation.

For some analyzers like Crush [11] that include multiple steps (e.g. it requires decompilation by Gigahorse [33] first) where not all steps require contract states, ETHERSPECT allows the analyzer to be split into multiple analyzer units and declare the dependencies among them to leverage the benefits of ETHERSPECT's analyzer scheduler.

The last thing to mention is the design for scalability of the analyzer scheduler. Without ETHERSPECT, the most straightforward approach to efficiently execute analyzers is data parallelism using a parallel computing framework such as multiprocessing pool provided by Python [34], which most related work such as SmartBugs [21, 20] use. However, such approaches leverage inter-process communication (IPC) services provided by operating

doi:10.6342/NTU202401985

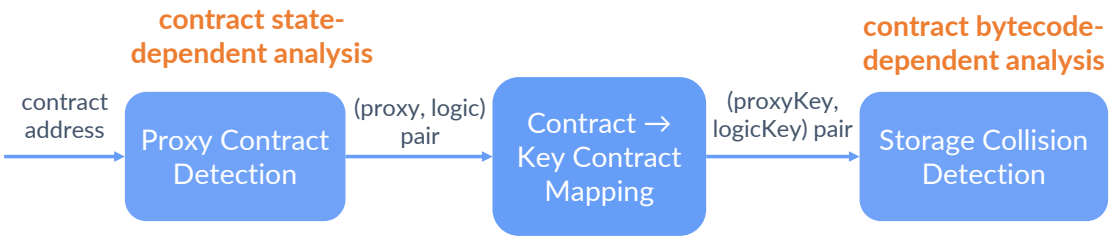


Figure 3.4: An example of key contract mapping.

systems, which do not scale out to multiple machines. EtherSpect's design is aware of that and can scale out to a cluster by using a database instead of IPC for synchronization.

3.5 Analyzer Executor

EtherSpect executes custom analyzers using an executor with the necessary setup to run the analyzer, and handles input and output. Since smart contract analyzers usually provide command-line interfaces (CLI) and often have library dependencies, for portability and flexibility, EtherSpect requires wrapping analyzers in containers and providing a small bootstrap script that reads an input file, launch the analyzer instance, and writing results to an output file. EtherSpect's analyzer scheduler puts inputs into input files and collects outputs from output files accordingly.

doi:10.6342/NTU202401985

3.6 Proxy Contract Detection Unit

The proxy contract detection unit is based on ProxyChecker [12]. It serves as a built-in analyzer and is executed to detect proxy contracts and find their logic contracts. Notably, the proxy-logic relationships are kept as a graph and serve as part of the dataset provided by ETHERSPECT, allowing additional analysis on this relationship graph.

doi:10.6342/NTU202401985

doi:10.6342/NTU202401985

Chapter 4 Implementation

This section describes the implementation details of the components of EtherSpect, including contract collector (Section 4.1), source code retriever (Section 4.3), proxy contract detection unit (Section 4.6), custom analyzer executor (Section 4.5) and analyzer scheduler (Section 4.4).

4.1 Contract Collector

The contract collector is based on ethereum-etl package [35], which parses raw transactions obtained from an Ethereum archive node into EVM execution traces, and then extracts contract address and bytecode from contract deployment and contract destruction operations discovered in the execution traces. These data are then inserted into the deployed contract list and the destructed contract list. Subsequently, the contract collector computes the Keccak256 hash of each contract's bytecode and inserts the bytecode into the *contract code* list using the bytecode hash as the key. The deployed contract list, the destructed contract list, and the *contract code* list are maintained as SQL tables in PostgreSQL, while the alive contract list is a SQL view with a reference block number parameter.

doi:10.6342/NTU202401985

4.2 Database Schema

ETHERSPECT defines several core database table schema that are used to organize data and analysis results in ETHERSPECT, as shown in SQL Data Definition Language in listing 4.1, 4.2 and 4.3. The `contracts` table stores information about each smart contract, such as whether being a proxy contract. The `contract_code` table stores information of each distinct smart contract bytecode, such as source code and analysis result of Slither and Crush. The implementation table stores information about proxy-logic contract pairs, such as collisions.

A key design of ETHERSPECT is the separation of *contracts* and *contract code* to skip redundant computation on duplicated bytecode (Figure 3.2). During the dependency-aware scheduling of analyzer instances, frequent mapping between the contracts and the contract_bytecode table in both directions is performed by the analyzer scheduler. This is achieved by the bytecode hash field of the contracts table, a foreign key referencing the contract_bytecode table, and the related_address field of the contract_bytecode table referencing backward to the contract table.

Based on these core schema, users of ETHERSPECT should be able to integrate various types of analyzers into the ETHERSPECT using the appropriate schema.

Listing 4.1: Schema of `contract` table

```
CREATE TABLE contracts (
    address CHAR(42) NOT NULL PRIMARY KEY,
    block_number INTEGER NOT NULL,
    block_timestamp TIMESTAMP NOT NULL,
    bytecode_hash CHAR(66) NOT NULL REFERENCES contract_code
    -- additional columns
);
```


doi:10.6342/NTU202401985

Listing 4.2: Schema of `contract_code` table

```
CREATE TABLE contract_code (
    bytecode_hash CHAR(66) NOT NULL PRIMARY KEY,
    related_address CHAR(42) NOT NULL,
    -- additional columns
);
```

Listing 4.3: Schema of `implementation` table

```
CREATE TABLE implementations (
    proxy_address CHAR(42) NOT NULL REFERENCES contracts,
    proxy_bytecode_hash CHAR(66) NOT NULL REFERENCES contract_code,
    logic_address CHAR(42) NOT NULL REFERENCES contracts,
    logic_bytecode_hash CHAR(66) NOT NULL REFERENCES contract_code,
    PRIMARY KEY (proxy_address, logic_address)
    -- additional columns
);
```

4.3 Source Code Retriever

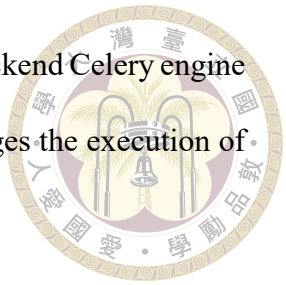
The source code retriever uses an API rate limiter based on NGINX reverse proxy in front of the Etherscan API to avoid being blocked from exhausting its request quota, which is 5 requests per second of a free-tier one. [32]

4.4 Analyzer Scheduler

The analyzer scheduler is based on Apache Airflow [36], a workflow orchestration framework. A backend PostgreSQL database is used to synchronize states throughout the EtherSpect pipeline. Analyzer instance states and results of each component in the pipeline are kept in the database. To scale out onto a cluster of worker nodes, we made the

doi:10.6342/NTU202401985

database a central service accessible by all nodes in the cluster. The backend Celery engine of Apache Airflow maintains a task queue on the database and manages the execution of analyzer instances on different nodes well.



4.5 Analyzer Executor

To provide the most flexibility, we open the implementation of executing analyzers to users of EtherSpect. In particular, developers freely design a database schema for keeping the result and write a Python script (called directed acyclic graph, or DAG, in Apache Airflow) to define how input/output is handled and how the analyzer launches in a container. We provide DAG templates that can be used to inherit and quickly craft a working DAG definition with minimal setup (e.g. database schema, input/output format processing, command to launch the analyzer). Recall that a key design of EtherSpect is that it differentiates contract and contract bytecode as different types of inputs. Either to take contract or contract bytecode as input should be defined in the DAG definition of the analyzer.

4.6 Proxy Contract Detection Unit

ProxyChecker is integrated into EtherSpect like a custom analyzer, wrapped in a container plus a DAG definition.

doi:10.6342/NTU202401985

Chapter 5 Finding Proxy Contracts

In this section, we use EtherSpect to discover all proxy contracts in Ethereum and detect collision issues.

Aligned with ProxyChecker, we define a proxy contract as a smart contract satisfying the following conditions:

1. Contains a DELEGATECALL instruction in its fallback function.
2. After being called targeting a non-existing function signature, the delegate call in the fallback function is triggered.
3. The delegate call perfectly forwards call data from the proxy contract to the logic contract.

When the target address of the delegate call is obtained from the bytecode of the proxy contract, we say it is a minimal proxy contract.

We try to answer the following research questions:

RQ1 How are proxy patterns used in recent years?

RQ2 Which are the most popular proxy contracts and logic contracts?

RQ3 How many proxy contracts have function collisions or storage collisions?

RQ4 What is the adoption of proxy standards?

RQ5 How frequently are proxy contracts upgraded?

doi:10.6342/NTU202401985

5.1 Setup

We run EtherSpect with ProxyChecker [12], Slither [9], Gigahorse [33] (a dependency of Crush) and Crush [11] as 4 custom analyzers to conduct the study. The input is a list of contract addresses, which goes to ProxyChecker and produces a graph of proxy contracts and associated logic contracts. Then for each proxy-logic pair in the graph, both contracts are mapped to the contract bytecode list. In all cases, the bytecode pair is passed to Crush after being decompiled by Gigahorse for finding collisions; specifically, if both contract bytecode of the proxy-logic pair has source code available, they are passed to Slither for finding collisions. An overview of the workflow is shown in Figure 5.1.

We select block number 19,823,648 (mined on May 8th, 2024) as the reference block to cover 37,662,073 contracts from the alive contract list (Section 3.2) which results in 1,259,362 distinct bytecode. The evaluation runs on a system with Ubuntu 24.04, AMD Ryzen 9 3900X 12-core 24-thread processor, running at 3.8 GHz each, and 64 GB of memory.

5.2 Proxy Contract Usage

To answer RQ1, we want to know the adoption of proxy patterns in smart contract applications over the years. Until May 8th, 2024, we identified 19,916,816 proxy contracts, and 20,255,454 proxy-logic pairs, the former accounting for 53% out of 37M alive contracts. Figure 5.2 presents proxy contracts deployed over years. It is shown that the majority of proxy contracts are minimal proxy [3] contracts, a special type of proxy contract that only contains about 20-30 instructions delegating function calls to logic contracts and is not compiled from source code. Such a proxy pattern is for cloning an existing contract

doi:10.6342/NTU202401985

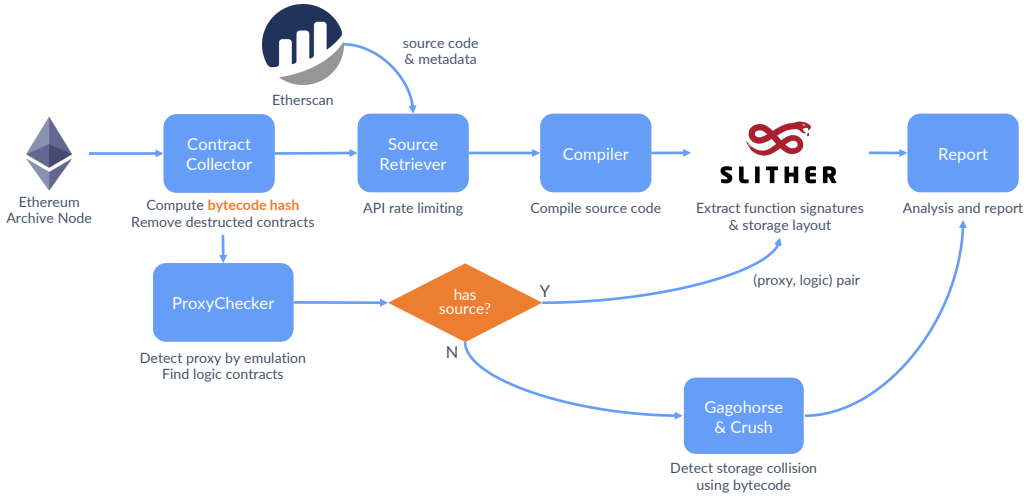


Figure 5.1: Proxy Contract Detection Flow

with a slight cost, though not providing upgradability. Among the rest most (2,135,951) non-minimal proxy contracts have verified source code on Etherscan; only 144,652 non-minimal proxy contracts have no available source code.

The turning point lies in 2020, after when proxy contracts abundantly emerged. In 2015-2020, only 2 million proxy contracts were deployed, but after 2020, more than 3 million proxy contracts were deployed each year. In 2023, the number went to 7 million in a single year. 2020 is also a year where several important standards about proxy contracts had been settled down [3, 4, 5, 6]. Since 2020, proxy contracts have been the mainstream. In 2022 and 2023 especially, 90% of deployed contracts are proxy contracts.

doi:10.6342/NTU202401985

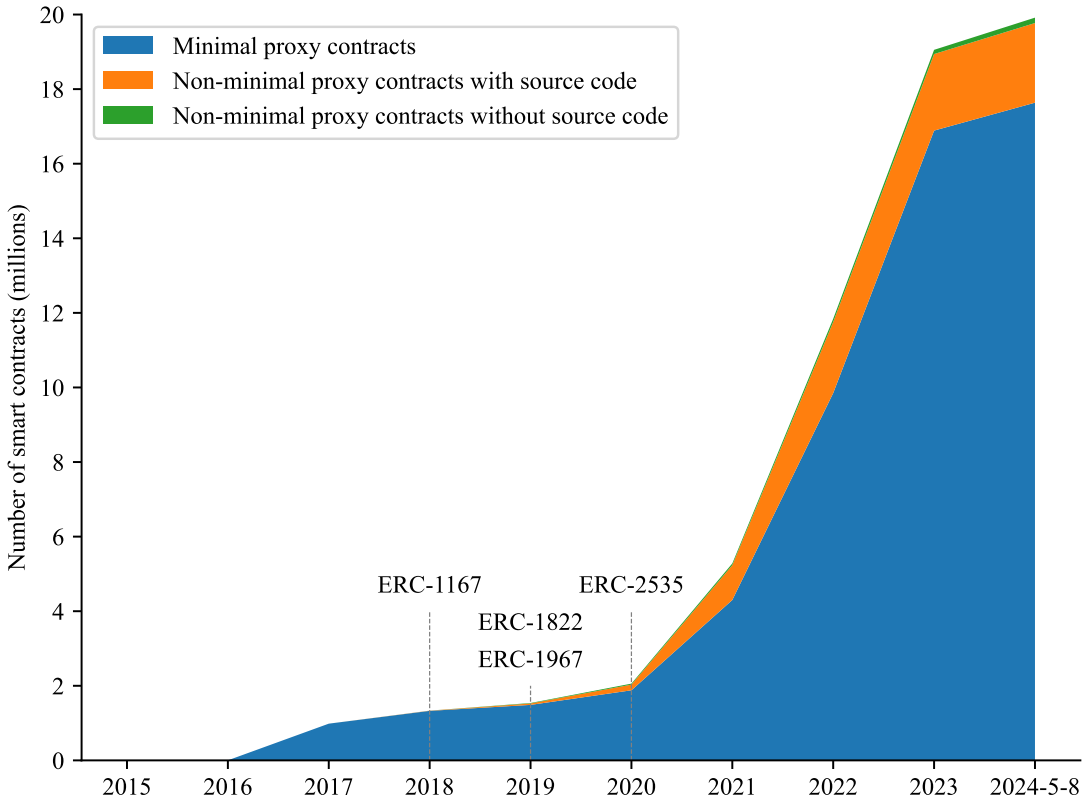


Figure 5.2: Accumulated number of proxy contracts identified until May 8th, 2024.

5.3 Top Duplicated and Top Referenced Proxy and Logic Contracts

To answer RQ2, we count the references of each logic contract by proxy contracts and count the number of replicated bytecodes for proxy contracts and logic contracts.

The top 3 most referenced logic contracts are CoinTool_App [1], XENTorrent [2], and

[1] https://etherscan.io/address/0x0de8bf93da2f7eecb3d9169422413a9bef4ef628#code
[2] https://etherscan.io/address/0x0a252663dbcc0b073063d6420a40319e438cfa59#code

doi:10.6342/NTU202401985

AuthenticatedProxy [3], having reference count ranging from 3.5M to 1.5M. The first 2 are related to the XEN token [37], while the third is part of the Wyvern Protocol, which we will discuss in the next paragraph.

Excluding minimal proxy contracts, the top 3 duplicated proxy contracts are OwnableDelegateProxy [4] (1.5M replicas), GnosisSafeProxy [5] (0.2M replicas), and WalletProxy [6] (40k replicas). The first always has AuthenticatedProxy as the logic contract (mentioned in the previous paragraph), and is a user-generated proxy contract in the Wyvern Protocol of Opensea NFT Marketplace [38], suggesting a high demand in the non-fungible token (NFT) market. The latter 2 are cryptocurrency wallets, an application of decentralized finance.

Logic contracts generally are not duplicated frequently (mostly less than 40), yet 2 outstanding ones have more than 10,000 duplicates [7] [8]. These contracts are created by CoinBase Commerce [9] [39], an onchain payment system.

In conclusion, the most popular proxy and logic contracts are essentially popular services where users are required to deploy proxy contracts by their protocol. These services include decentralized finance applications especially wallets and tokens, and non-fungible token applications.

5.4 Function Collisions and Storage Collisions

To answer RQ3, we run Slither [9] and Crush [11] on the proxy-logic contract pairs we found.

[3] https://etherscan.io/address/0xf9e266af4bca5890e2781812cc6a6e89495a79f2#code
[4] https://etherscan.io/address/0x0a08e6058eaaa847a1adb55b0a69b8469ea5a5b3#code
[5] https://etherscan.io/address/0xdab5dc22350f9a6aff03cf3d9341aad0ba42d2a6#code
[6] https://etherscan.io/address/0xf29351130fb53547ac594a6cd30655e87976e346#code
[7] https://etherscan.io/address/0x9695756bec02db42311472a9c2a7f8e9c23f6bf3#code
[8] https://etherscan.io/address/0x3dd928b1f5e7999b87f6d9cf2440994aae5fe816#code
[9] https://etherscan.io/address/0x881d4032abe4188e2237efcd27ab435e81fc6bb1#code

doi:10.6342/NTU202401985

We follow the definitions of function collisions and storage collisions of these tools. In Slither, function collisions are the intersection of 4-byte function signatures of the proxy contract and the logic contract, no matter if the function name collides or not; and storage collisions are storage variables that occupy the same storage slots but have different names or different types in the proxy contract than in the logic contract. In Crush, since they are bytecode-based, with no knowledge about the names, they defined storage collision as storage slots that have different types in the proxy contract than in the logic contract.

Minimal proxy contracts neither access storage slots nor have functions so are excluded, resulting in 2,347,319 non-minimal proxy-logic pairs. Slither only runs on a subset of these contracts where both proxy and logic contracts have source code (2,095,662 pairs, 89%), while Crush can cover all of them. To speed up the analysis process, we run Slither and Crush on the contract bytecode list mapped from each proxy-logic pair, greatly reducing the time needed for evaluation.

As a result, 1,585,428 function collisions (5,167 distinct bytecode) and 25,691 storage collisions (456 distinct bytecode) are found by Slither, as shown in Table 5.1. The vast majority of function collisions are introduced by the pair OwnableDelegateProxy (proxy) and AuthenticatedProxy (logic), contributing 1,545,768 function collisions.

By manual inspection, we found quite a lot of function collisions resulting from the auto-generated getter function of storage variables. That is a feature of the Solidity compiler to automatically generate a public getter function of the variable name for every public storage variable. When a proxy contract shares the same public storage variables with the logic contract, though not considered a storage collision, the auto-generated getter functions are always deemed as function collisions. In the example of the OwnableDelegateProxy and AuthenticatedProxy pair, they share 2 public variables on the same stor-

doi:10.6342/NTU202401985

Collision Type	# Contracts	Percentage	# Contract Bytecode	Percentage
function collision (Slither)	1,585,428	7.83%	5,167	3.66%
storage collision (Slither)	25,691	0.13%	456	0.32%
storage collision (Crush)	1,555,812	7.68%	1,762	1.25%

Table 5.1: The number of collisions found out of 20M proxy-logic contract pairs.

age slots: `address implementation` and `address upgradeabilityOwner`. In this case, the same storage layout is not a storage collision, but the generated getter function `implementation()` and `upgradeabilityOwner()` collides.

As for storage collisions, we found that many of them are false positives where storage variables only slightly differ in their name, such as adding an underscore. Some others are mapping-type variables, which are not exploitable due to their distinct layout (which seems randomized slots) in the storage.

On the other hand, Crush found 1,555,812 storage collisions. The result is significantly divergent from Slither. We guess that, since Crush infers storage variable types from decompiling bytecode, more inaccuracy is yielded than source code-based Slither.

5.5 Proxy Standard Adoptions

To answer RQ4, we calculate the adoption rate of the currently recommended ERC standards about proxy contracts [3, 4, 5, 6].

The mechanism in ProxyChecker checking for the compliance of proxy standards is by determining if specific storage slots defined by the standards are used to store the logic contract's address, see Table 5.2 and Figure 5.3. The only exceptions are ERC-1167 [3], which is identified by matching the exact pattern defined by the standard in the bytecode, and OpenZeppelin's transparent proxy [29], which is identified by keywords in the source code.

We see that the ERC-1167 Minimal Proxy Contract [3] is the most popular among the

doi:10.6342/NTU202401985

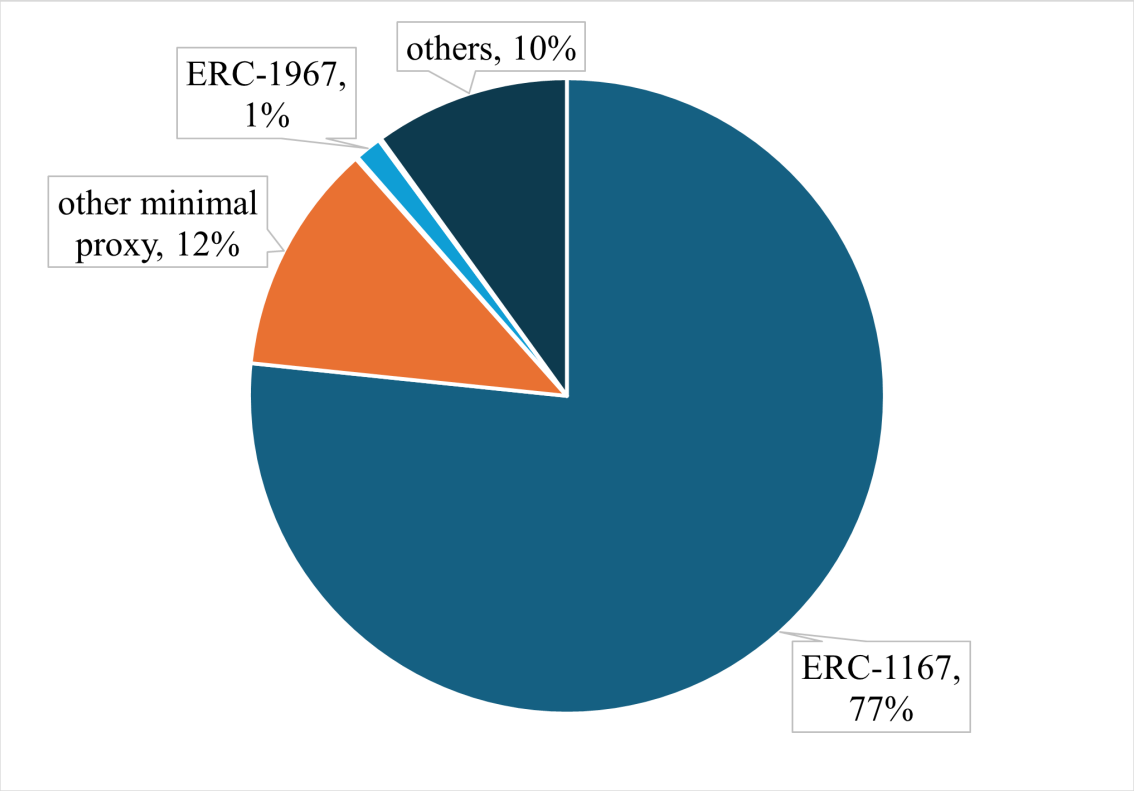


Figure 5.3: The number of proxy contracts following specific standards. Refer to table 5.2 for small values that are ignored in this chart.

Proxy Type	**# Contracts**	**Percentage**	**Location of the logic contract's address**
ERC-1167 [3]	15,285,105	76.7%	In the bytecode offset 10-29
Non-standard minimal proxy	2,351,108	11.8%	In the bytecode
ERC-1822 [4]	22,721	0.11%	Slot = `Keccak-256("PROXIABLE")`
ERC-1967 [5]	272,742	1.37%	Slot = `Keccak-256("eip1967 .proxy.implementation")`
OpenZeppelin's transparent proxy [29]	25,941	0.13%	Follows ERC-1967
ERC-2535 [6]	365	0.002%	Slot = `Keccak-256("diamond .standard.diamond.storage")`
Others	1,984,775	9.95%	Other places, not limited to storage slots

Table 5.2: The number of proxy contracts following specific standards, and their percentage out of the total number of proxy contracts identified (19.9M).

doi:10.6342/NTU202401985

standards, accounting for 76.7% among total alive contracts, implying a strong demand for functionality cloning.

The second most popular is ERC-1967 Proxy Storage Slots [5] (12% out of non-minimal proxy), a standard that intends to eliminate storage collisions by enforcing proxy contracts to use special slots that are never allocated by the logic contract. This suggests that not a few developers have been aware of the problem of storage collisions. This standard has a well-recognized implementation by OpenZeppelin called transparent proxy [29], which further eliminates function collisions by implementing minimal functionality in the proxy contract, leaving only upgrade functions. We found that 10% out of the ERC-1967 contracts use that implementation.

The ERC-1822 Universal Upgradable Proxy [4] Contracts have fewer usage. It is a pattern that moves upgrade logic from proxy contracts to logic contracts to reduce gas consumption. However, this pattern complicates the contract design and produces the potential of bricking the proxy contract if not upgraded carefully to a correct logic contract [4, 40]. Since upgrade logic is now in logic contracts, such action may be irreversible, which may be one of the reasons such a pattern is not as popular.

The last standard to mention is ERC-2535 Diamond Proxy Contract [6], a relatively novel proxy pattern introduced in 2020. It extends the proxy pattern by supporting multiple upgradable logic contracts, providing modular design in complicated smart contract applications. ProxyChecker fails to detect diamond proxy contracts due to its limitation [12], but we can still discover some of them by scanning the usage of storage slots (Table 5.2).

There are still considerable (10%) proxy contracts that do not comply with any of the recommended standards, leaving a dangerous space for collisions and exploits.

doi:10.6342/NTU202401985

5.6 Upgrades on Proxy Contracts

To answer RQ5, we count the total upgrade events of non-minimal proxy contracts in history. Surprisingly, most (97.7%) non-minimal proxy contracts are never upgraded to a new version of logic contracts. Only an insignificant number of 53,061 have been upgraded. That implies that a lot of developers have chosen proxy patterns but did not perform any upgrades, leaving potential vulnerabilities and bugs. That being said, from the perspective of decentralization, some applications avoid frequent upgrades without users' strong consensus to not compromise users' trust in their transparency and consistency, especially for applications involving valuable assets.

doi:10.6342/NTU202401985

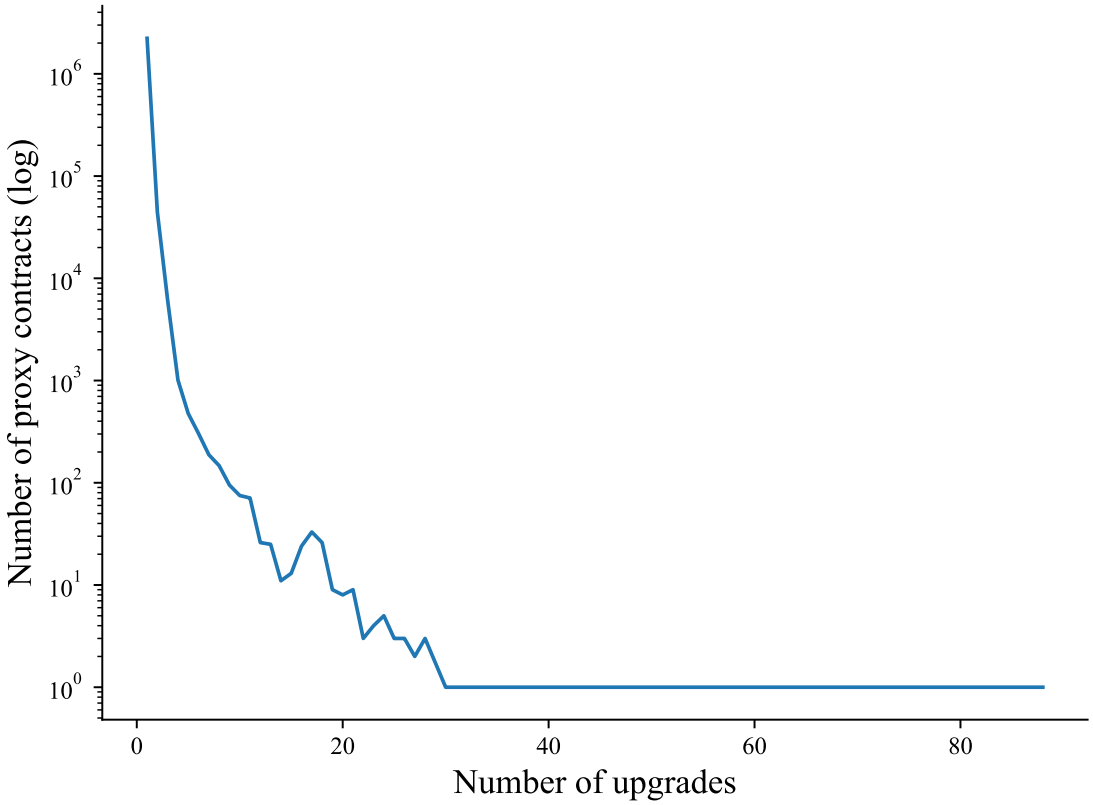


Figure 5.4: Number of upgrades for logic contracts in log scale. Most non-minimal proxy contracts (97.7%) have not upgraded to a newer version of the logic contract.

doi:10.6342/NTU202401985

doi:10.6342/NTU202401985

Chapter 6 Evaluting ETHERSPECT

In this section, we evaluate ETHERSPECT's effectiveness, performance, extensibility, and scalability.

Specifically, we try to answer the following research questions:

RQ1 How many manual efforts are saved by ETHERSPECT?

RQ2 Can Etherspect easily integrate third-party analyzers?

RQ3 What is the latency and throughput of each analyzer in ETHERSPECT?

RQ4 What is the scheduling overhead of Etherspect?

RQ5 Can Etherspect scale out to multiple machines?

RQ6 How much time does Etherspect save to analyze all proxy contracts compared to a naïve approach?

6.1 Automation & Extensibility

We try to answer RQ1 and RQ2 by discussion.

ETHERSPECT intends to be automated and extensible by: First, ETHERSPECT provides a complete up-to-date dataset with necessary preprocessing that be directly used to conduct an evaluation. Second, ETHERSPECT handles the execution of analyzers, includ-

doi:10.6342/NTU202401985

ing dependency-aware scheduling, failure handling, and timeout, in a parallel computing setup.

In summary, users of EtherSpect only need to write the following code and let the rest for EtherSpect to manage:

1. A database schema based on EtherSpect's core schemas (Section 4.2) to store results of the analyzer.

2. Handle input and output in database format.

3. Handle input and output format in the filesystem, if the analyzer requires.

4. A Docker image for running the analyzer.

5. A startup script to launch the analyzer inside a container.

As we have integrated ProxyChecker, Slither, Gigahorse, and Crush, the integration of these analyzers takes an average of 300 lines of code, showing EtherSpect's capability to plug in custom analyzers.

6.2 Latency and Throughput

To answer RQ3, we measure the average latency and throughput of each analyzer running in EtherSpect. Latency is defined as the total time required to finish analyzing a contract. We measure latency by averaging the time consumed per contract in each batch, with scheduling overhead amortized into the latency of each contract. Throughput, on the other hand, is defined as the number of contracts that can be processed within a period. We measure throughput by counting the total number of contracts that are taken by the analyzer for execution within a time interval. We show the results in table 6.1.

doi:10.6342/NTU202401985

	Latency (sec)	1 / Latency (contract per sec)	Throughput (contracts per sec)	Speedup
ProxyChecker [12]	0.13	7.7	178.5	23.2
Slither [9]	2	0.5	12	24
Gigahorse [33]	26.2	0.04	0.88	22
Crush [11]	242	0.004	0.12	30

Table 6.1: Latencies & throughputs of analyzers used in our study, running on EtherSpect on our hardware setup

6.3 Overhead

The analyzer scheduler introduces overhead, including database accesses for analyzer state synchronization and input/output management, and the decision process of the analyzer scheduler. We measure these overheads to answer RQ4. We set up ProxyChecker as the single analyzer, let the batch size be 100,000, and ran for 3.5 hours, resulting in 25 batches being processed. We observed an average input/output access overhead of 10.22s, and a scheduling decision overhead of 1.08s, accounting for 2% and 0.2% of the total execution time, respectively.

6.4 Scalability

To learn how EtherSpect can scale out to multiple machines and answer RQ5, we add another machine and form a 2-node cluster. The machine has an AMD Ryzen Threadripper 2990WX 32-Core 64-thread Processor, 64GB RAM, running Ubuntu 24.04. We compared Crush's throughput after adding the additional computing node, and show the result in table 6.2. A speedup of 2.8x with one more machine exhibits EtherSpect's scalability (with an additional 64 cores to the existing 24 cores, the maximum theoretical speedup can be estimated to (24+64)/64 = 3.67x). This implies that without manually partitioning the dataset, EtherSpect does well in distributing the workload onto multiple

doi:10.6342/NTU202401985

	Throughput (contracts per sec)
single node	0.12
2 nodes	0.34
speedup	2.8x

Table 6.2: Throughputs of Crush running on a single machine and a 2-node cluster, by dynamically distributing data using ETHERSPECT's analyzer scheduler

machines.

We agree that randomly splitting the dataset into 2 partitions in a ratio based on the machine's computing power, i.e. their number of logical cores, can achieve a similar throughput speedup without additional overhead. Yet, we argue that it costs human effort to split the dataset, and the partition strategy may not yield optimal performance, especially when the dataset includes some odd long-running instances that exceed the timeout and block other instances, where a dynamic scheduler may utilize resources more efficiently.

6.5 Total Time Savings by ETHERSPECT

To show how ETHERSPECT can complete a large-scale analysis more efficiently than a naive approach and answer RQ6, we take the large-scale discovery of proxy contracts as a case study (§5).

Assuming the analysis pipeline is run on a machine with the same configuration (24 logical cores), with a dataset of 37M alive contracts. If we do not deduplicate identical bytecode from the 37M dataset, by multiplying the average execution time (the reciprocal of throughput) obtained from table 6.1 of ProxyChecker (5.6ms), Slither (83ms), Gigahorse (1.14s) and Crush (8.2s), the total execution time is estimated to 2228 days. Moreover, the lack of a dependency-aware analyzer scheduler leads to manual efforts to maintain a correct execution order of the analyzers and be involved after each analysis phase for result data processing.

doi:10.6342/NTU202401985

	Estimated Time Required
naive approach	2228 days
with ETHERSPECT	25 days

Table 6.3: Time required for a comprehensive analysis using a naive approach or with ETHERSPECT

With ETHERSPECT, with automatically and strategically skipping duplicated bytecode by ETHERSPECT analyzer scheduler (3.4) on Slither, Gigahorse, and Crush phases, we make it possible to finish within 25 days, with a small scheduling overhead of 2% as mentioned in Section 6.3. In detail, 37M alive contracts only have 1.2M distinct bytecode, and while ProxyChecker discovers 20M proxy-logic pairs, there are only 54k pairs of distinct bytecode. Not to mention, ETHERSPECT's scaling-out capability implies an even faster evaluation with sufficient hardware resources.

doi:10.6342/NTU202401985

doi:10.6342/NTU202401985

Chapter 7 Discussion

7.1 Limitations of ETHERSPECT

In this section, the limitations of ETHERSPECT and the large-scale analysis methodology taken by ETHERSPECT are discussed.

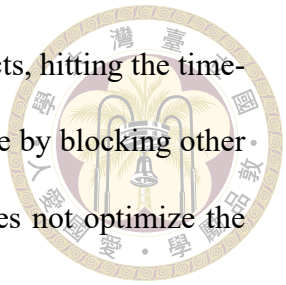
7.1.1 Smart Contract Data and Knowledge

ETHERSPECT is a smart contract analysis platform that provides a preprocessed smart contract dataset aimed for large-scale analysis, and a scheduler that executes smart contract analyzers. As a smart contract dataset, ETHERSPECT only provides bytecode, source code, and transactions, excluding high-level knowledge about smart contracts, such as their purposes (being a DeFi application or an NFT), value transfers, and interaction history, which are useful in conducting application-level studies [41]. Additionally, in security research, a vulnerability list is helpful, like what SmartBugs [20] and Web3Bugs [42] provides. For future work, such information can be gathered from online resources such as Etherscan [13], and DeFiHackLabs [43].

7.1.2 Scheduling Algorithm and Resource Utilization

In a parallel computing environment, task distribution among threads and workers can greatly affect overall performance. For example, from our experience in large-scale

doi:10.6342/NTU202401985



evaluation, analyzers often do not halt when analyzing certain contracts, hitting the time-out and resulting in failures. This greatly impacts overall performance by blocking other instances from computing resources. Yet, EtherSpect currently does not optimize the timeout but instead leaves it as a manual configuration.

In addition, EtherSpect fetches input and writes output in batches to reduce I/O overhead. The batch size is also a determinable parameter to balance I/O and execution time and improve overall performance, while it is set manually now.

Currently, EtherSpect's analyzer scheduler does not use complex scheduling logic, but only a simple first-come-first-serve algorithm. There should be a space for research on designing a good scheduling algorithm. Possible directions are pursuing maximal resource utilization or minimal latency, predicting time-consuming instances, and being aware of analyzer dependencies.

7.1.3 Scaling Out

Scaling EtherSpect to numerous machines may present several challenges.

First, the current architecture relies on a centralized database (PostgreSQL) for data synchronization and a network file system (NFSv4) for runtime data and logs, which may become communication bottlenecks as the system scales. A potential solution is to replace these components with distributed databases and distributed file systems, both of which are well-established technologies [44, 45]. The primary challenges will likely involve configuring and tuning these systems to accommodate our specific workload.

Second, as the number of workers increases, effectively distributing tasks to them becomes more critical to performance and more challenging, as discussed in Section 7.1.2.

doi:10.6342/NTU202401985

7.2 Future Work of ETHERSPECT

In this section, future work on ETHERSPECT is discussed.

7.2.1 Online Vulnerability Detection

Given the stringent security requirements of smart contract applications, particularly those involving valuable assets, online vulnerability detection remains an active area of research [46]. In this context, ETHERSPECT is sufficiently automated to operate independently on-chain for the detection of proxy contracts and collision vulnerabilities, provided that ProxyChecker, Slither, Crush, or other security analyzers are integrated.

7.2.2 Modifying the Implementation of Analyzers

To further improve the efficiency of analyzers or to adapt the execution environment, sometimes the implementation of analyzers needs to be modified. Essentially, Etherspect was designed with generality in mind to allow analyzers to be integrated with minimal changes. However, ETHERSPECT also aims at efficiency in large-scale analysis, which leads to an emphasis on supporting performance optimizations.

For example, we have considered caching highly repetitive or time-consuming data (e.g., intermediate representations) to reduce re-computation. To this end, adding a general caching API (e.g., *get* and *put* operations like a key-value store) in Etherspect may help to leverage Etherspect's scheduling mechanism for such needs.

In general, ETHERSPECT enables developers to either integrate analyzers with minimal integration costs or adapt and optimize performance on them.

doi:10.6342/NTU202401985

7.3 Limitations of Our Study on Proxy Contracts

Limitations to our approach in studying all proxy contracts in Ethereum lie in limited coverage and overapproximation.

First, ProxyChecker overlooks diamond proxy contracts [6] in that they reject a random function signature in the call data during emulation [12]. By examining storage slots that common diamond proxy contracts may use, we did not find many of these contracts in Ethereum, suggesting that they might have less popularity for now.

Second, we found so many function collisions and storage collisions that most of them do not seem to be exploitable. Though we did not modify the detection algorithm of Slither and Crush, we argue there should be more sophisticated techniques to filter out those false positives. Crush [11] tried to investigate exploitability on storage collisions using symbolic execution, yet we do not find related work on exploitability of function collisions.

7.4 Best Practices for Developing Proxy Contracts

The increasing prevalence of proxy contract standards and security-oriented implementations suggests a trend in the smart contract developer community to recognize collision issues and attempt to avoid them. Besides the standards established by Ethereum community [3, 4, 5, 6], one of the most famous proxy contract implementations is Transparent Proxy developed by OpenZeppelin [29]. Such implementation completely avoids function collisions and storage collisions by implementing minimal functionality and using only randomized storage slot numbers in the proxy contract.

doi:10.6342/NTU202401985

Chapter 8 Related Work

We list related work that has built smart contract datasets (either manually prepared or automatically generated) and analysis frameworks in this section and that have studied proxy contracts.

The most complete smart contract dataset is the Ethereum public dataset on Google BigQuery [17, 18, 19], containing a smart contract list directly extracted from an Ethereum archive node automatically on a daily basis. It provides bytecode, related transactions, balances, and some other information about token and wallet contracts for decentralized finance, but does not identify duplicated bytecode which is important for large-scale analysis. Another dataset is Smart Contract Sanctuary [15], which aims to provide smart contract source code, which is a manually maintained GitHub repository and outdated for at least half a year at the time of writing. A likewise recognized dataset is SmartBugs Wild Dataset [22, 20], which targets data for analysis use ranging from bytecode, deduplicated bytecode to source code. Its further extension [14] provides bytecode skeletons, trimmed bytecode that provide the same functionality. Their preprocessing on the dataset is intended to be run manually, which is not ideal from an automation perspective. Su et al. [47] built an automated extract-transform-load (ETL) pipeline that extracts smart contract interactions into a graph database, but their system is not integrated with smart contract analyzers.

There are also some well-established smart contract analysis frameworks. Smart-

doi:10.6342/NTU202401985

Bugs [20] and its 2.0 version [21] are frameworks suitable for large-scale evaluation of smart contract analyzers as presented by the authors [22, 14]. They provide a preprocessed dataset and an analyzer scheduler that enables parallel computation. However, it is limited in scalability for distributed computing environments and complex analysis that includes mutually dependent analysis steps. SolidiFi [48] also aimed at automatically and systematically evaluating multiple tools. Besides these, there are more related work that, though less automated, targets systematic approaches to evaluate smart contract analysis tools [30, 31, 42, 49].

Finally, studies on proxy contracts appeared in recent years. Etherscan provides a simple mechanism to identify proxy contracts by scanning for DELEGATECALL instructions in the contract bytecode, with a limited accuracy acknowledged by their official [50]. Slither is a smart contract analysis tool that builds in a proxy detection unit that works by searching for related keywords in the source code. This method still suffers from low accuracy. USCHunt [10] further extended Slither to cover proxy contracts more accurately, but the coverage of their study was quite limited (to only contracts with source code). Crush [11] is the state-of-the-art at the time of writing, which explores all storage collisions by analyzing bytecode and transactions. However, their approach requires bytecode decompilation and is thus inefficient.

doi:10.6342/NTU202401985

Chapter 9 Conclusion

In conclusion, we filled a gap in a comprehensive study on all proxy contracts in Ethereum by building EtherSpect for efficient and automated analysis. We gained some understanding of the status of proxy contracts and their collision issues, including finding out the popularity of proxy patterns, the adoption of proxy standards, and identifying 1,585,428 function collisions and 25,691 storage collisions. Furthermore, EtherSpect was proven to make large-scale analysis more efficient, relieving efforts of future research on smart contracts.

doi:10.6342/NTU202401985

doi:10.6342/NTU202401985

References

[1] Xiaofan Li, Jin Yang, Jiaqi Chen, Yuzhe Tang, and Xing Gao. 2024. Characterizing Ethereum Upgradable Smart Contracts and Their Security Implications. In *Proceedings of the ACM on Web Conference 2024 (WWW '24)*. Association for Computing Machinery, New York, NY, USA, 1847–1858. https://doi.org/10.1145/3589334.3645640

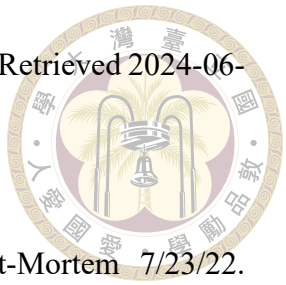
[2] Jorge Izquierdo and Manuel Araoz. 2018. ERC-897: DelegateProxy. Retrieved 2024-06-16 from https://eips.ethereum.org/EIPS/eip-897

[3] Peter Murray, Nate Welch, and Joe Messerman. 2018. EIP-1167: Minimal Proxy Contract. Retrieved 2024-06-16 from https://eips.ethereum.org/EIPS/eip-1167

[4] Gabriel Barros and Patrick Gallagher. 2019. ERC-1822: Universal Upgradeable Proxy Standard (UUPS). Retrieved 2024-06-16 from https://eips.ethereum.org/EIPS/eip-1822

[5] Santiago Palladino, Francisco Giordano, and Hadrien Croubois. 2019. ERC-1967: Proxy Storage Slots. Retrieved 2024-06-16 from https://eips.ethereum.org/EIPS/eip-1967

doi:10.6342/NTU202401985



[6] Nick Mudge. 2020. ERC-2535: Diamonds, Multi-Facet Proxy. Retrieved 2024-06-16 from https://eips.ethereum.org/EIPS/eip-2535

[7] Audius Music. 2022. Audius Governance Takeover Post-Mortem 7/23/22. Retrieved 2024-06-16 from https://blog.audius.co/article/audius-governance-takeover-post-mortem-7-23-22

[8] Shaurya Malwa. 2022. How Attackers Stole Around $1.1M Worth of Tokens From Decentralized Music Project Audius. Retrieved 2024-06-16 from https://www.coindesk.com/tech/2022/07/25/how-attackers-stole-around-11m-worth-of-tokens-from-decentralized-music-project-audius/

[9] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 8–15. https://doi.org/10.1109/WETSEB.2019.00008

[10] William E Bodell III, Sajad Meisami, and Yue Duan. 2023. Proxy Hunting: Understanding and Characterizing Proxy-based Upgradeable Smart Contracts in Blockchains. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 1829–1846. https://www.usenix.org/conference/usenixsecurity23/presentation/bodell

[11] Nicola Ruaro, Fabio Gritti, Robert McLaughlin, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna. 2024. Not your Type! Detecting Storage Collision Vulnerabilities in Ethereum Smart Contracts. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. San Diego, USA.

doi:10.6342/NTU202401985

[12] 朱玟嶧. 2022. 探討以太坊代理合約所引發的安全問題. Master's thesis. 國立臺灣大學. https://doi.org/10.6342/NTU202203210

[13] Etherscan. 2024. Etherscan - The Ethereum Blockchain Explorer. Retrieved 2024-06-16 from https://etherscan.io/

[14] Monika di Angelo, Thomas Durieux, João F Ferreira, and Gernot Salzer. 2024. Evolution of automated weakness detection in Ethereum bytecode: a comprehensive study. *Empirical Software Engineering* 29, 2 (2024), 41. https://doi.org/10.1007/s10664-023-10414-8

[15] Martin Ortner and Shayan Eskandari. 2018. Smart Contract Sanctuary. Retrieved 2024-06-16 from https://github.com/tintinweb/smart-contract-sanctuary

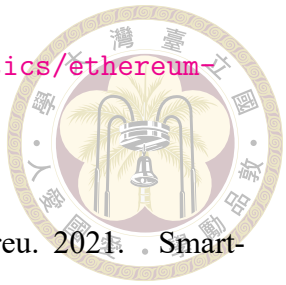
[16] Etherscan. 2024. Etherscan Verified Contracts. Retrieved 2024-06-16 from https://etherscan.io/contractsVerified

[17] Google BigQuery. 2024. Ethereum Cryptocurrency. Retrieved 2024-06-16 from https://console.cloud.google.com/marketplace/product/ethereum/crypto-ethereum-blockchain

[18] Allen Day and Evgeny Medvedev. 2018. Ethereum in BigQuery: a Public Dataset for smart contract analytics | Google Cloud Blog. Retrieved 2024-06-16 from https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics

[19] Evgeny Medvedev and Allen Day. 2018. Ethereum in BigQuery: how we built this dataset | Google Cloud Blog. Retrieved 2024-06-16 from

doi:10.6342/NTU202401985



https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-how-we-built-dataset

[20] João F. Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. 2021. SmartBugs: a framework to analyze solidity smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1349 – 1352. https://doi.org/10.1145/3324884.3415298

[21] Monika di Angelo, Thomas Durieux, João F. Ferreira, and Gernot Salzer. 2023. SmartBugs 2.0: An Execution Framework for Weakness Detection in Ethereum Smart Contracts. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2102–2105. https://doi.org/10.1109/ASE56229.2023.00060

[22] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 530–541. https://doi.org/10.1145/3377811.3380364

[23] Ethereum. 2024. What is Ethereum? Retrieved 2024-06-16 from https://ethereum.org/en/what-is-ethereum/

[24] Solidity Team. 2023. Solidity Programming Language. Retrieved 2024-06-16 from https://soliditylang.org/

[25] Vyper community. 2024. Vyper. Retrieved 2024-06-16 from https://vyperlang.org/

doi:10.6342/NTU202401985

[26] Morteza Zakeri-Nasrabadi, Saeed Parsa, Mohammad Ramezani, Chanchal Roy, and Masoud Ekhtiarzadeh. 2023. A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges. *Journal of Systems and Software* 204 (2023), 111796. https://doi.org/10.1016/j.jss.2023.111796

[27] Tinchoabbate. 2024. Beware of the proxy: learn how to exploit function clashing - Security - OpenZeppelin Forum. Retrieved 2024-06-16 from https://forum.openzeppelin.com/t/beware-of-the-proxy-learn-how-to-exploit-function-clashing/1070

[28] Christof Ferreira Torres, Mathis Steichen, and Radu State. 2019. The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1591–1607. https://www.usenix.org/conference/usenixsecurity19/presentation/ferreira

[29] OpenZeppelin. 2024. Transparent Proxy | OpenZeppelin Docs. Retrieved 2024-06-16 from https://docs.openzeppelin.com/contracts/4.x/api/proxy#transparent_proxy

[30] Heidelinde Rameder, Monika di Angelo, and Gernot Salzer. 2022. Review of Automated Vulnerability Analysis of Smart Contracts on Ethereum. *Frontiers in Blockchain* 5 (2022). https://doi.org/10.3389/fbloc.2022.814977

[31] Satpal Singh Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur, and Heung-No Lee. 2022. Ethereum Smart Contract Analysis Tools: A Systematic Review. *IEEE*

doi:10.6342/NTU202401985

Access 10 (2022), 57037–57062. https://doi.org/10.1109/ACCESS.2022.3169902

[32] Etherscan. 2024. Etherscan APIs. Retrieved 2024-06-16 from https://etherscan.io/apis

[33] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: Thorough, Declarative Decompilation of Smart Contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1176–1186. https://doi.org/10.1109/ICSE.2019.00120

[34] Python. 2024. multiprocessing —Process-based parallelism. Retrieved 2024-06-16 from https://docs.python.org/3/library/multiprocessing.html

[35] Evgeny Medvedev and Blockchain ETL. 2018. Ethereum ETL. https://github.com/blockchain-etl/ethereum-etl

[36] Apache Airflow. 2024. Apache Airflow. Retrieved 2024-05-19 from https://airflow.apache.org/

[37] XEN Crypto. 2024. XEN Crypto. Retrieved 2024-06-16 from https://www.xencrypto.io/

[38] Wyvern. 2023. Wyvern Protocol. Retrieved 2024-06-16 from https://wyvernprotocol.com/

[39] Coinbase. 2024. A new standard in global crypto payments - Coinbase. Retrieved 2024-06-16 from https://www.coinbase.com/commerce

[40] Noxx. 2024. Smart Contract Patterns: The Proxy. Retrieved 2024-06-16 from https://noxx.substack.com/p/smart-contract-patterns-the-proxy

doi:10.6342/NTU202401985

[41] Liyi Zhou, Xihan Xiong, Jens Ernstberger, Stefanos Chaliasos, Zhipeng Wang, Ye Wang, Kaihua Qin, Roger Wattenhofer, Dawn Song, and Arthur Gervais. 2023. SoK: Decentralized Finance (DeFi) Attacks. In *2023 IEEE Symposium on Security and Privacy (SP)*. 2444–2461. https://doi.org/10.1109/SP46215.2023.10179435

[42] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. 2023. Demystifying Exploitable Bugs in Smart Contracts. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, 615 – 627. https://doi.org/10.1109/ICSE48619.2023.00061

[43] SunWeb3Sec. 2022. SunWeb3Sec/DeFiHackLabs: Reproduce DeFi hacked incidents using Foundry. Retrieved 2024-06-16 from https://github.com/SunWeb3Sec/DeFiHackLabs

[44] Ali Davoudian, Liu Chen, and Mengchi Liu. 2018. A Survey on NoSQL Stores. *ACM Computing Surveys (CSUR)* 51, 2, Article 40 (apr 2018), 43 pages. https://doi.org/10.1145/3158661

[45] Peter Macko and Jason Hennessey. 2022. Survey of Distributed File System Design Choices. *ACM Transactions on Storage (TOS)* 18, 1, Article 4 (mar 2022), 34 pages. https://doi.org/10.1145/3465405

[46] Rong Cao, Ting Chen, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, Yuxing Tang, Xiaodong Lin, and Xiaosong Zhang. 2020. SODA: A Generic Online Detection Framework for Smart Contracts. In *Proceedings of the 27th Network and Distributed System Security Symposium (NDSS)*. 1–17.

doi:10.6342/NTU202401985

[47] Voon Hou Su, Sourav Sen Gupta, and Arijit Khan. 2022. Automating ETL and Mining of Ethereum Blockchain Network. In *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining (WSDM '22)*. Association for Computing Machinery, New York, NY, USA, 1581 – 1584. https://doi.org/10.1145/3488560.3502187

[48] Asem Ghaleb and Karthik Pattabiraman. 2020. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 415–427. https://doi.org/10.1145/3395363.3397385

[49] Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Benjamin Livshits. 2024. Smart Contract and DeFi Security Tools: Do They Meet the Needs of Practitioners?. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 60, 13 pages. https://doi.org/10.1145/3597503.3623302

[50] Etherscan. 2024. Proxy Contracts. Retrieved 2024-06-16 from https://info.etherscan.com/what-is-proxy-contract/

doi:10.6342/NTU202401985