Computer Modeling in Engineering & Sciences

Tech Science Press

DOI: 10.32604/cmes.2022.021562

check for updates

ARTICLE

Analyzing Ethereum Smart Contract Vulnerabilities at Scale Based on Inter-Contract Dependency

Qiuyun Lyu[1], Chenhao Ma[1], Yanzhao Shen[2], Shaopeng Jiao[3], Yipeng Sun[1] and Liqin Hu[1,*]

[1]School of Cyberspace, Hangzhou Dianzi University, Hangzhou, 310018, China

[2]Research and Development Department, Shandong Institute of Blockchain, Jinan, 250102, China

[3]Security Team, Shanghai Anshi Network Technology, Ltd., Shanghai, 200233, China

*Corresponding Author: Liqin Hu. Email: huliqin@hdu.edu.cn

Received: 21 January 2022 Accepted: 09 June 2022

ABSTRACT

Smart contracts running on public blockchains are permissionless and decentralized, attracting both developers and malicious participants. Ethereum, the world's largest decentralized application platform on which more than 40 million smart contracts are running, is frequently challenged by smart contract vulnerabilities. What's worse, since the homogeneity of a wide range of smart contracts and the increase in inter-contract dependencies, a vulnerability in a certain smart contract could affect a large number of other contracts in Ethereum. However, little is known about how vulnerable contracts affect other on-chain contracts and which contracts can be affected. Thus, we first present the contract dependency graph (CDG) to perform a vulnerability analysis for Ethereum smart contracts, where CDG characterizes inter-contract dependencies formed by DELEGATECALL-type internal transaction in Ethereum. Then, three generic definitions of security violations against CDG are given for finding respective potential victim contracts affected by different types of vulnerable contracts. Further, we construct the CDG with 195,247 smart contracts active in the latest blocks of the Ethereum and verify the above security violations against CDG by detecting three representative known vulnerabilities. Compared to previous large-scale vulnerability analysis, our analysis scheme marks potential victim contracts that can be affected by different types of vulnerable contracts, and identify their possible risks based on the type of security violation actually occurring. The analysis results show that the proportion of potential victim contracts reaches 14.7%, far more than that of corresponding vulnerable contracts (less than 0.02%) in CDG.

KEYWORDS

Smart contract vulnerability; smart contract homogeneity; contract dependency graph; automated analysis

1 Introduction

With the rapid development of blockchain technology, the application of blockchains has extended from simple decentralized point-to-point payment to decentralized applications in various fields, in which the emergence and popularization of smart contracts play an important role. A smart contract, a Turing-complete program running on the blockchain, enables users to develop different decentralized applications that meet their own needs [1], such as new tokens [2–4], marketplaces for

digital assets [5], and so on. Those running on public blockchains have the characteristics of openness and decentralization allowing arbitrary users to participate [6].

However, these characteristics also attract many malicious participants, who only need a general blockchain account and a few low-cost transactions to exploit vulnerabilities in any on-chain contract, especially, Ethereum; The world's largest decentralized application platform on which more than 40 million smart contracts are running according to the Ethereum public dataset on Google BigQuery [7] are running, is frequently challenged by smart contract vulnerabilities. What's worse, vulnerabilities in a certain contract could affect a large number of other on-chain contracts or even the entire Ethereum ecosystem, due to the widely revealed homogeneity of smart contracts [8,9] and the increase in inter-contract connections [9,10]. A real-world case is the Parity Wallet Bug [11–13]. In less than half a year from July to November 2017, the Parity Wallet project suffered two attacks initiated by unknown Ethereum accounts, resulting in significant financial losses to nearly 600 wallet contracts. These vulnerabilities are not in wallet contracts themselves, but in the shared library contract [14] on which they depend. The case shows that as multiple contracts work together to complete a function, they will share the risk of vulnerabilities.

Although several existing works [15–18] have done large-scale analysis on the Ethereum smart contract vulnerabilities, two problems still have not been considered. On the one hand, a vulnerable contract could put other on-chain contracts that depend on it at risk, just like the real-world case above, yet they are unable to specify which on-chain contracts would be affected. On the other hand, due to the inter-contract connections, whether a contract is vulnerability-free rests not only with its own bytecode but also with the bytecode of contracts on which it depends [17]. Besides, their analyses stayed at the state of Ethereum smart contract vulnerabilities before 2019, but little is known about how their states have changed in recent years. In fact, an increasing number of complex decentralized applications require the cooperation of multiple contracts nowadays, such as dynamic libraries [19], version control [20] and duplicate proxies of smart contracts [21]. These contracts form an inter-contract dependency through a special type of internal transaction, namely DELEGATECALL, but existing tools or schemes do not combine these interconnected contracts in the process of their vulnerability analysis. Therefore, a new large-scale analysis of current Ethereum smart contract vulnerabilities based on inter-contract dependency is urgently needed.

In this paper, we combine contract automated analysis with graph analysis of Ethereum to explore the impact of vulnerable contracts on other contracts. We first present the concept of contract dependency graph (CDG), a directed acyclic graph with contracts as nodes and DELEGATECALL-type internal transactions as edges, to capture inter-contract dependencies in Ethereum. Subsequently, we formally define three categories of security violations against CDG on which different types of vulnerable smart contracts can be checked for finding potential victims affected by them. Then, we perform large-scale analysis which focuses on three specific known vulnerabilities to check their security violations against the CDG that was constructed with the latest Ethereum internal transaction data (from January 2, 2020 to August 10, 2021). Our analysis shows that 38 vulnerable contracts make 28,786 contracts become potential victims, in total after considering the Ethereum smart contract homogeneity. Compared with previous large-scale analyses that simply mark vulnerable contracts with known vulnerabilities, our analysis for the first time marks a new class of contracts that do not have known vulnerabilities but may become victims of these vulnerable contracts. Our scheme can be extended to more types of known vulnerabilities considering that different types of vulnerabilities pose different risks to potential victim contracts. In summary, we make the following three contributions to this paper:

- A Contract Dependency Graph (CDG) is defined to characterize inter-contract dependencies in Ethereum, smart contract homogeneity and DELEGATECALL sequence are considered in it. Further, we identify three classes of security violations against CDG to formally describe how different vulnerable contracts affect other contracts in CDG.
- A CDG-based vulnerabilities analysis for Ethereum smart contracts is designed and implemented on the latest Ethereum. In addition to vulnerable contracts, our analysis scheme also identifies potential victim contracts that are not caught by previous works, since their lack of the organization for multiple contracts based on inter-contract dependencies.
- Our analysis shows that a few vulnerable contracts create hundreds of times more potential victimized contracts in Ethereum, similar to what happens with some real-world vulnerability exploits. Compared to previous large-scale analysis, four observations worthy of attention about current Ethereum smart contract vulnerabilities landscape are mentioned.

The remainder of this paper is organized as follows: Section 2 briefly introduces Ethereum smart contract and related work in recent years. Section 3 describes the definition of contract dependency graph and three categories of security violations against it. In Section 4, the design and implementation of a large-scale analysis with real-world datasets are presented. Section 5 shows the result of our analysis in detail and compares it with the work of other researchers. The final section summarizes this paper and presents a preliminary discussion on the future directions.

2 Preliminary and Related Work

In this section, we give an introduction to Ethereum smart contract, its execution model and different types of inter-contract interactions. Then we briefly describe other researchers' work in the field of smart contract vulnerabilities automated analysis and graph analysis of Ethereum.

2.1 Ethereum Smart Contract and Execution Model

Ethereum smart contract, a special type of Ethereum account, is uniquely identified by a 160-bit address on blockchain. It stores bytecode usually compiled by Solidity, which is immutable and repeatable. In other words, the bytecode of all smart contracts cannot be modified once deployed, and those identified by different addresses may have exactly the same bytecode [6]. Besides, Ethereum smart contract is an event-driven code contract with state attributes that are permanently recorded on Ethereum [22].

The bytecode of each smart contract is interpreted and executed by the Ethereum Virtual Machine (EVM); the execution model specifies how the system state is altered given a series of opcodes and a small tuple of environmental data. To keep our notation close to the formal specification of Ethereum [23], we use μ to denote the EVM machine state with memory μ_m and stack μ_s. $\sigma[a]$ is used to denote the account state of address a with storage $\sigma[a]_S$. Furthermore, we use I to refer to the execution environment data, where I_d is the data field of the transaction which can be malicious user-controlled inputs. As of July 29, 2021, there are a total of 145 active opcodes [24], each of which has different semantics to manipulate the data in all types of the memory above (μ_m, μ_s, $\sigma[a]_S$ and I_d), e.g., CALLDATACOPY copies part of the I_d to μ_m, and SLOAD reads a 256-bit value from $\sigma[a]_S$ to μ_s. In particular, the opcodes to perform a write operation on $\sigma[a]_S$ are defined as state-changing operations.

2.2 Internal Transaction for Inter-Contract Interaction

In Ethereum, the execution of EVM is triggered by the transaction, a message call from one account address to another. According to the account type of the transaction caller, a transaction can be classified as an external transaction and an internal one. The former is sent by an externally owned account (EOA), and a smart contract account sends the latter. An internal transaction cannot be performed directly, but is attached to at least one external transaction. Ethereum offers four types of internal transactions for inter-contract interaction: CALL, CALLCODE, DELEGATECALL and STATICCALL. Table 1 shows the differences between all types of message calls, by defining the caller contract address as u, the callee contract address as v, the caller field of external transaction that results in the execution of current message call as *tx.origin*, and the caller field of the current message call as *msg.sender*. Specifically, the function of STATICCALL is equivalent to CALL, except it disallows any state-changing operations during the call (and its subcalls, if present).

Table 1: Differences among CALL, CALLCODE, DELEGATECALL and STATICCALL

Call Type	*msg.sender*	Execution Context
CALL	u	$\sigma[v]_s$
CALLCODE	u	$\sigma[u]_s$
DELEGATECALL	*tx.origin*	$\sigma[u]_s$
STATICCALL	u	$\sigma[v]_s$

Noted that a DELEGATECALL-type internal transaction makes the caller contract dependent on the callee contract at a bytecode or a storage level. On the one hand, the caller contract can call methods in the callee contract through DELEGATECALL in its own context, denoted as bytecode-level inter-contract dependency, which is similar to a traditional binary program executing third-party library functions. On the other hand, DELEGATECALL preserves the execution environment data of the original call (e.g., *tx.origin*), and state-changing operations of the callee contract acts on the storage of the caller contract, denoted as storage-level inter-contract dependency. For consistency, we consider a DELEGATECALL-type internal transaction between two contracts to be equivalent to having an inter-contract dependency between them in the remainder of this paper.

2.3 Smart Contract Vulnerabilities Automated Analysis

Automated analysis of smart contracts, whose main motivation is to detect various types of known contract vulnerabilities, has been a field of active research for the past few years. Krupp et al. [15] proposed a method to identify vulnerabilities caused by transaction sequences and automatically generate corresponding exploitations. In the same vein, Nikolic et al. [16] presented three types of tracking vulnerabilities, each of which results from tracing vulnerable transaction sequences of a contract over its lifetime. So et al. [25] extended the considered transaction sequences by guiding symbolic execution with a language model for effectively hunting vulnerable ones. A few automated analysis tools support inter-contract analysis, including Manitcre [26], Mythril [27] and ETHBMC [17], where Manitcre supports such analysis in a non-automatic way that requires a multi-contract environment with the storage value at a certain block for each contract account. Both Mythril and ETHBMC offer an on-chain analysis that relies on a running Ethereum archive node to automatically load other contracts' bytecodes from the blockchain when needed.

2.4 Graph Analysis of Ethereum

Graph analysis empowers us to better understand the Ethereum ecosystem [10]. Unlike Bitcoin, which only has money transfer activities between users, the majority of Ethereum's activities revolve around smart contracts, such as smart contract creation and invocation. Kiffer et al. [9] focused on the activities of smart contract creation and interaction between EOAs and contract accounts that occurred in the first 5,000,000 blocks (January 30, 2018) of Ethereum. Their research reveals the fact that most contracts are copies of other contracts. Chen et al. [10] used Ethereum's transaction from July 30, 2015 to November 01, 2018 to construct three types of graphs depicting money transfers, contract creation and contract invocation, which were used to solve three types of security issues: anomaly detection, attack forensics and deanonymization.

Compared to all those approaches, which focus on either automated analysis for a single contract, or building relationship graph for the Ethereum ecosystem, our work combines contract automated analysis with Ethereum graph analysis in order to explore the impact of vulnerable contracts on other contracts, given that different contracts are not isolated but rather depend on each other in the Ethereum ecosystem. Besides, the latest Ethereum internal transaction data (from January 02, 2020 to August 10, 2021) is collected as support for our work, which more realistically reflects the current state of the Ethereum ecosystem.

3 Contract Dependency Graph

In this section, we first propose the concept of contract dependency graph to characterize inter-contract dependencies on Ethereum. Several properties of CDG that are not directly derived from the definition are discussed in Section 3.1. Then, we define three types of security violations against CDG for our analysis scheme to find respective potential victim contracts affected by different types of vulnerable contracts in Section 3.2.

3.1 Definition of CDG

There are sufficient incentives to disperse complex functional logic to multiple different contracts, including gas cost reduction [11], code reuse [28], contract version iteration [20] and so on. One contract can invoke methods provided by other on-chain contracts through internal transactions. Due to the complex environment of blockchain, Ethereum provides a variety of invocation methods, the most special of which is DELEGATECALL, which makes the caller contract dependent on the callee one at a bytecode or a storage level (see Section 2.2). To characterize such activities throughout the Ethereum smart contract ecosystem, we propose the concept of CDG.

Definition 3.1. $CDG = (V, E)$, where V is a set of nodes, E is a set of edges. Each $v_i \in V$ is a smart contract represented by a tuple $v = (A, C)$, where A is the 160-bit address used to uniquely identify the contract, C is the bytecode of contract $\sigma\,[A]$. E is a set of ordered pairs of nodes, $E = \{(v^i, v^j) | v^i, v^j \in V\}$, where a directed edge (v^i, v^j) from v^i to v^j indicates there is at least one DELEGATECALL-type internal transaction whose caller contract address equals to v^i_A and callee one equals to v^j_A.

Different from the Ethereum smart contract graph constructed by existing work [10] without considering the problem of contract homogeneity [8,9], $v^i, v^j \in V(v^i_A \neq v^j_A)$ are judged to be identical if their bytecodes are identical when constructing the CDG. This practice is adopted by many other works [15,18] to extract bytecode-unique contracts by ignoring the duplicate ones. Besides, $(v^i, v^m), (v^j, v^n) \in E(v^i_A \neq v^j_A, v^m_A \neq v^n_A)$ with both the same tail node and the same head node will be merged, since the presence of multiple edges is disallowed in CDG. According to the above description, CDG is the modeling of multiple non-isolated contracts, where there are no nodes with exactly the

same bytecode, and each node is connected by inter-contract dependencies. Fig. 1 is an example demonstrating our method of CDG construction, in which contract "0x70e . . . 489", "0xb35 . . . 321" and "0x5d5 . . . f73" have exactly the same bytecode, as do contract "0xc99 . . . 5c5" and "0xa65 . . . bc3". As a result, the constructed CDG has only three nodes connected by two edges, but covers all the inter-contract dependencies between bytecode-unique contracts. For convenience, we use the terms bytecode-unique contract and node interchangeably in the remainder of this paper.

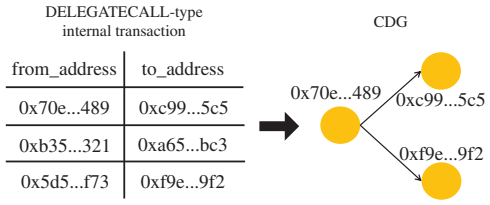

Figure 1: An example demonstrating our method of CDG construction

In the following, we focus on the connectivity of CDG and prove that it is a directed acyclic graph (DAG) from the perspective of gas mechanism.

We assume the $CDG = (V, E)$ is a non-DAG, equivalent to the statement:

$$\exists v \in V, P(v, v) \neq \emptyset, \tag{1}$$

where $P(v, v)$ indicates that contract v calls its own methods by the way of loop call. Correspondingly, a calling contract's own methods is always converted into a conditional JUMPI that leads to the starting program counter address of the callee method, denoted as an internal call. The following gas analysis proves that each $P(v, v)$ will be replaced by an internal call that invokes the same function (i.e., execute the same bytecode), yet with lower gas cost.

According to the Ethereum's yellow paper [23], the general gas cost function for all EVM opcodes, C, is defined as:

$$C(\sigma, \mu, I) \equiv C_{mem}(\mu_i^{'}) - C_{mem}(\mu_i) + C_w, \tag{2}$$

where σ and μ represent the World State and the Machine State (see Section 2.1), and μ_i is defined as the maximum number of words of active memory, and w is one of the EVM opcodes. To compare the gas cost of a loop call with that of an internal call, we define the gas cost of a loop call as C_{loop}, another one as $C_{internal}$. Moreover, the gas cost of the function F called by both of them as C_F.

$$\begin{aligned} C_{internal} &= C(\sigma, \mu, I_b[n:m]) \\ &= \sum_{t=n}^{m} \left(C_{mem}\left(\mu_i^{'t} t\right) - C_{mem}\left(\mu_i^{t}\right) + C_{I_b[t]} \right) \\ &= C_{mem}(\mu_i^{'m}) - C_{mem}(\mu_i^{n}) + C_F + C_{other} \end{aligned} \tag{3}$$

where $I_b[n:m]$ indicates the byte array calling F in the way of an internal call. Based on the fee schedule stipulated in the Ethereum's yellow paper, we have

$$C_{other} \approx C_{PUSH4} + C_{EQ} + C_{PUSH2} + C_{JUMPI} + C_{JUMPDEST} = 20 \tag{4}$$

As $I_b[j:k]$ indicates the byte array calling F in the way of a loop call, we have

$$\begin{aligned} C_{loop} &= C(\sigma, \mu, I_b[j:k]) \\ &= \sum_{t=j}^{k} \left(C_{mem}\left(\mu_i^{'t}\right) - C_{mem}\left(\mu_i^{t}\right) + C_{I_b[t]} \right) \\ &= C_{mem}(\mu_i^{'k}) - C_{mem}(\mu_i^{j}) + C_F + C_{other}^{'} \end{aligned} \tag{5}$$

Similarly, by defining the path length of $P(v, v)$ as n, we have

$$C_{other}^{'} \approx nC_{call} > 700n(n \geq 2) \tag{6}$$

Since both the loop call and the internal call invoke the same function which performs the same state transition for EVM, we have

$$C_{mem}(\mu_i^{'m}) - C_{mem}(\mu_i^{n}) = C_{mem}(\mu_i^{'k}) - C_{mem}(\mu_i^{j}) \tag{7}$$

So far we have proved that $C_{loop} > C_{internal}$. Thus, none of $v \in V$ has a loop $P(v, v)$ to itself that will be replaced by internal calls with lower gas costs.

3.2 Security Violations against CDG

With the concept of CDG, we define and distinguish three categories of security violations against CDG based on the location and characteristics of vulnerable contracts that have different types of known vulnerabilities, in order to describe how vulnerable contracts affect other contracts in CDG. It is worth mentioning that for a node $v \in V$, all the nodes that depend on v are equivalent to $\{v' | P(v', v) \neq \emptyset, v' \in V\}$.

3.2.1 Deviation Violation

Deviation means that a malicious smart contract controlled by an attacker is unauthorizedly added into CDG, replacing its equivalent that was originally called by the vulnerable contract. Specifically, a caller contract in CDG that is called normally cannot be controlled by arbitrary Ethereum accounts, but can be controlled by the contract owner who has access to control [29]. However, a vulnerable contract that allows malicious Ethereum accounts to bypass the access control exposes the CDG to the risk of Deviation. Given the above insights, we formally define the Deviation violation against CDG as follows.

Definition 3.2. $CDG = (V, E)$ suffers Deviation at node $v \in V$ whose out-degree is non-zero, if v is a vulnerable node by which an attacker can replace node $\{m \in v' | (v, v') \in E, v' \in V\}$ with attacker-control contracts. All the nodes that depend on it becomes potential victims of this security violation, suffering from code injection by malicious contracts.

Taking Code-Injection [15] for example, a vulnerable node detected with such a known vulnerability can lead to the security violation of Deviation in CDG. More specifically, once the node exists the problematic uses of DELEGATECALL opcode where the second stack argument $\mu_s[1]$ denoting the callee of the internal call is an attacker-controlled smart contract account address B, the smart contract $\sigma[B]$ is considered as the new callee replacing its counterpart.

3.2.2 Interruption Violation

The Interruption means that there is at least one node that can be removed unauthorizedly from CDG causing the internal transactions from callers be interrupted at the current node. Further, once a callee contract in CDG is removed by an attacker, its caller contract is unable to call its function to get the expected result. Therefore, we give the formal definition of the Interruption violation against CDG as follows.

Definition 3.3. $CDG = (V, E)$ suffers Interruption at node $v \in V$ whose in-degree is non-zero, if v is a vulnerable node which can be unauthorizedly destructed by an attacker. All the nodes that depend on v become potential victims of such a security violation, suffering from out of service.

A real-world case, for example, is Suicidal vulnerability [25] where a contract can be killed by any arbitrary Ethereum account. More specifically, once a node in CDG exists, the execution of reachable SELFDESTRUCT opcodes where the only stack argument $\mu_s[0]$ representing the address to which all remaining funds of this contract will be transferred is either an attacker-controlled address or an arbitrary one. The vulnerable contract can be removed at any time with its balance transferred to the address specified by $\mu_s[0]$, and all contracts that depend on it will suffer from out of service, such as Locked Ether [30].

3.2.3 Betrayal Violation

Different from the above security violations, Betrayal (see Definition 3.4.) does not make any structural change to CDG, but allows the same vulnerabilities to propagate along a path in CDG from vulnerable contracts to other contracts. Specifically, if a callee contract has callable bug codes or logical errors an attacker has the opportunity to let its caller contracts execute the same bug codes or make the same logical errors in their own context. Both caller and callee contract are victims of such an attack. The formal definition of the Betrayal security violation against CDG is as follows.

Definition 3.4. $CDG = (V, E)$ suffers Betrayal at node $v \in V$ if v is a vulnerable node having callable bug codes or logical errors. For the nodes that depend on v, an attacker utilizes call sequences starting from them to execute the same bug codes or logical errors in their own context. Neither being removed nor replaced, the original victim node v behaves like a betrayer, creating new victims in nodes that depend on it. v together with all the nodes that depend on it become potential victims of this security violation.

The security violation of Betrayal can be extended to various known vulnerabilities, including Integer Overflow, Steal-Ether [30], and so on, since exploiting vulnerabilities in an on-chain smart contract, rather than replacing or breaking it, is more common in Ethereum, according to previous research [15,17]. For example, Steal-Ether vulnerability [17] allows an attacker to transfer Ether from node v to attacker-controlled addresses or arbitrary addresses unauthorizedly. More specifically, once a node in CDG exists, the bug code that causes unrestricted Ether transfers using CALL opcodes with the following propositions:

- the second stack argument $\mu_s[1]$ representing the beneficiary of the transaction is an attacker-controlled address or an arbitrary address;
- the third stack argument $\mu_s[2]$ denoting the number of value that will be transferred by this call is non-zero;
- reaching a normal stopping opcode, including STOP and RETURN.

The attacker would continue to transfer Ether in nodes that depend on v by utilizing their DELEGATECALL to the bug code of v. In other words, the potential victims of Betrayal will suffer from the same vulnerability as the vulnerable contract.

The three classed of security violations against CDG are intuitive, either replacing, removing or exploiting contracts in CDG. Many known vulnerabilities can be converted into one of three security violations after being assigned exact checkable properties. However, some of known vulnerabilities cannot translate into corresponding security violations against CDG. Take Re-Entrancy [30] as an example, an attacker-controlled contract can make loop calls to attack the vulnerable contract itself, but cannot further exploit, replace or remove it, thereby affecting other contracts in CDG. In Section 4, we focus our attention on three specific known vulnerabilities, including Code-Injection, Suicidal vulnerability and Steal-Ether to check of their security violations against CDG in our vulnerability analysis scheme based on inter-contract dependencies.

Noting that both CDG and related security violations against it are not targeted at a specific type of business scenario, such as decentralized finance applications, since only the inter-contract dependencies among contracts are preserved and considered in our CDG-based analysis scheme. Therefore, by constructing the CDG with specific types of smart contracts and accurately defining the checkable properties of specific business vulnerabilities, it can be extended to a variety of specific scenarios.

Finally, it should be noted that the current definition of CDG cannot effectively reflect the execution path within contracts and the degree of dependency between contracts.

4 Large-Scale Analysis Based on CDG

We implement a large-scale analysis for the entire Ethereum smart contract ecosystem to find three types of security violations against CDG. As shown in Fig. 2, the process of our large-scale analysis consists of three phases, which are described in the following sections. The first phase, data acquisition (Section 4.1), shows the source and scope of real-world Ethereum data. Then, the details of CDG construction are explained in the second phase (Section 4.2). The third phase (Section 4.3) conducts our security violations detection algorithm for CDG to get vulnerability reports of contracts active in the Ethereum ecosystem.

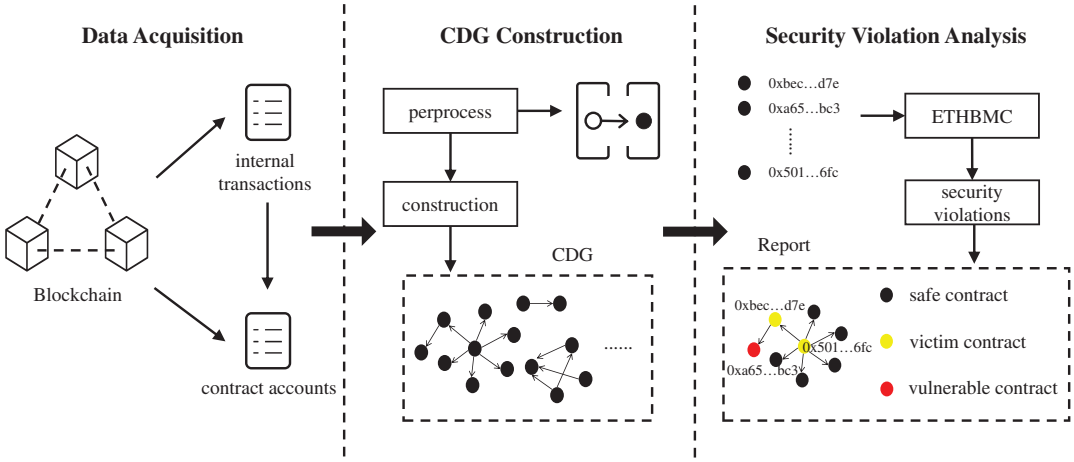

Figure 2: An overview of our approach for finding security violations against CDG

4.1 Data Acquisition

Different from external transactions directly obtained from the blockchain, the acquisition of internal transactions needs the Ethereum client to log all of the successfully-executed operations that modify the internal state of the EVM. For the correctness and completeness of on-chain data, the best approach is to retrieve these data directly from an Ethereum node with the tracing enabled and fully synced, but we do not have such a node available. Therefore, we obtain internal transactions and smart contract accounts from Google BigQuery [7], an Ethereum public dataset, also used by the authors of ETHBMC [17] as an alternative. Specifically, we query all DELEGATECALL-type internal transactions covering a time period from January 30, 2018 through August 10, 2021, and then analyze the trend of their daily number over time, as shown in Fig. 3. Since the number of DELEGATECALL-type internal transactions increased significantly after January 02, 2020 (over 100,000 per day), we collect all relevant data since then (block numbers ranging from 9,200,000 to 13,000,000), including 214,414,657 DELEGATECALL-type internal and 3,465,796 smart contract accounts involved.

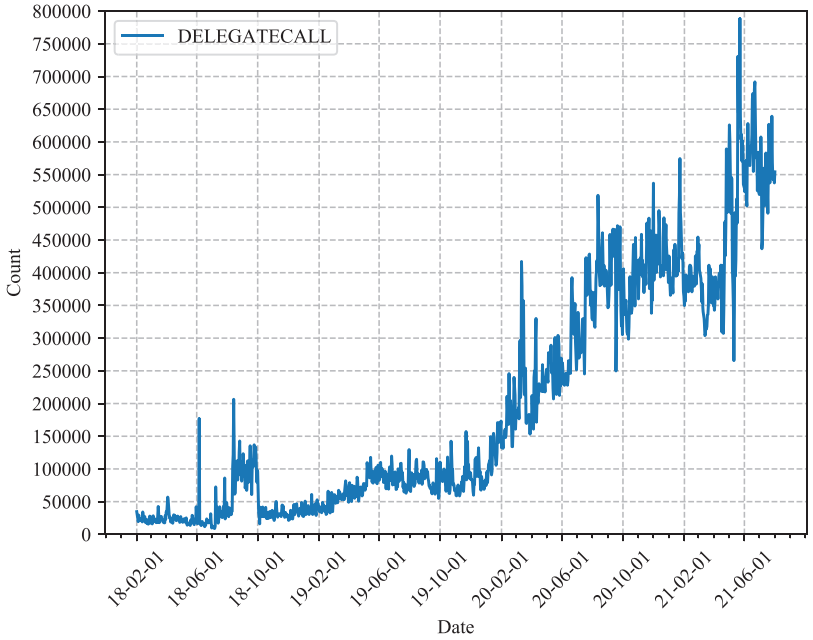

Figure 3: The number of DELEGATECALL-type internal transactions over time

4.2 CDG Construction

In the second phase, we first preprocess the above transactions to filter out the inactive inter-contract dependencies in the Ethereum ecosystem. Subsequently, we construct a CDG based on the definition given in Section 3.1, and an algorithm calculating attribute values for nodes in CDG is given.

4.2.1 Transaction Preprocessing

We first preprocess all internal transactions by the following conditions: (a) extracting unique internal transactions with addresses of caller contract and callee contract, and (b) the cumulative number of a unique one is greater than or equal to 10 (“active” to some extent). The preprocessing result shows that only 5.63% (195,247) of all contracts involved in internal transactions we collected

are active. The reason we only keep active contracts is that the degree distribution of most contract-related activities in Ethereum follows the power law [9,10]. Therefore, active contracts generally hold more Ether and are more likely to be targeted by attackers. Besides, a mapping from bytecodes to 160-bit addresses is built at this stage, used to obtain multiple Ethereum contract account addresses corresponding to the C attribute of a CDG node, considering that contracts identified by different addresses may be copies of the same bytecode. In our work, the number of bytecode-unique contracts acting as CDG nodes accounts for only 7% (13,682) of all active contracts, and the rest are copies of them.

4.2.2 Construction Stage

Based on the definition previously proposed to characterize inter-contract dependencies in the Ethereum ecosystem, we build a CDG with unique DELEGATECALL-type internal transactions obtained from the data acquisition phase as edges, and bytecode-unique contracts involved as nodes. Such a CDG models the inter-contract dependencies among contracts that are active between the 9,200,000th and 13,000,000th block of Ethereum.

Observation of the CDG shows that it consists of multiple disconnected subgraphs, indicating that each inter-contract dependency is limited to a group of contracts. In other words, a subgraph of CDG denotes a collection of all on-chain contracts that share the risk of vulnerabilities in Ethereum ecosystem. Fig. 4 presents a statistics of different types of subgraphs based on the number of nodes in CDG defining the SG_i as a subgraph of CDG whose number of nodes is equal to i. Each subgraph consists of a number of nodes (at least 2, up to 1,905 in our work) with inter-contract dependencies; the number of nodes is used as the basic unit of subsequent vulnerability analysis. Visualization of $SG_{>6}$ in CDG is given in Fig. 5.

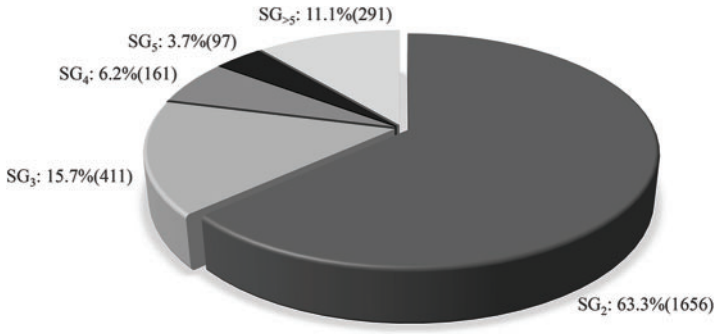

Figure 4: Statistics of different types of subgraphs based on the number of nodes in CDG

Note that we can find call sequences consisting of sequential DELEGATECALL in CDG, which only exist in seven subgraphs with a maximum length of 3. Fig. 6 shows these subgraphs and specially marks the nodes involved in the DELEGATECALL sequences (the graph structure of four subgraphs is exactly the same, one of them is reserved).

After building the CDG, we use Algorithm 1: to calculate a dependency hash table H for a CDG subgraph $SG = (V, E)$. For a bytecode-unique contract $v = (A, C) \in V$, all other contracts that depend on it in subgraph SG will store in $H(v)$. We calculate the $H(v)$ in the order of topological sorting (lines 1–3), ensuring the $H(v)$ of caller contracts always calculated before that of callee contracts. Since each node has a $H(v)$ corresponding to it, the $H(v)$ of the node whose in-degree is not zero is only related to its predecessor nodes (lines 6–11).

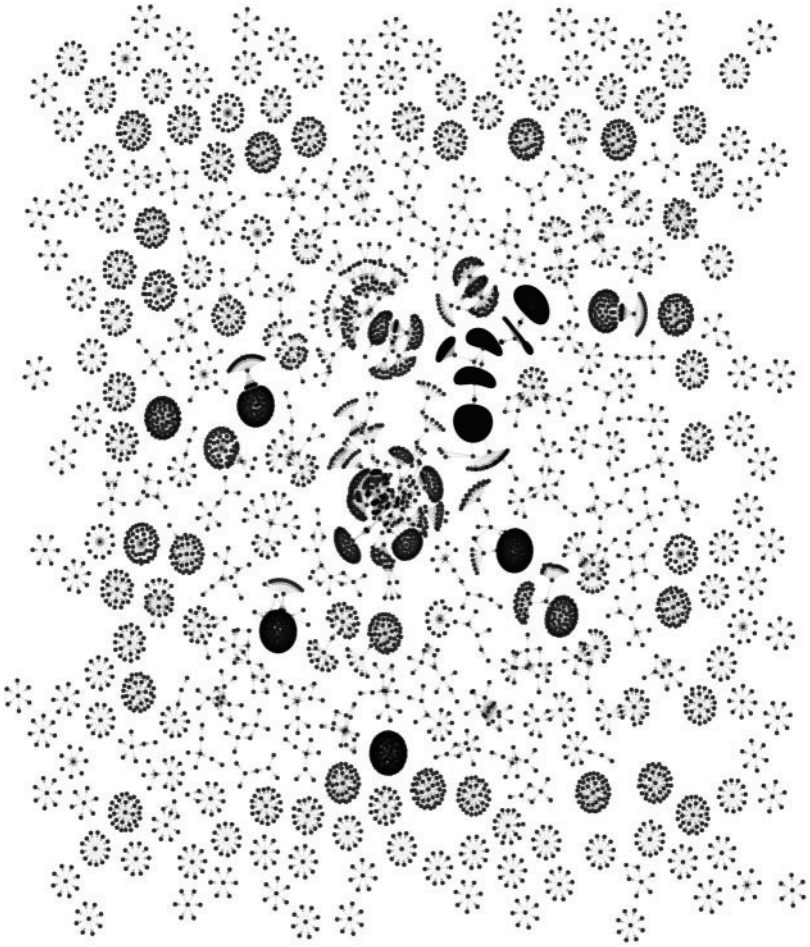

Figure 5: Visualization of CDG (for the ease of illustration, only $SG_{>6}$ are chosen)

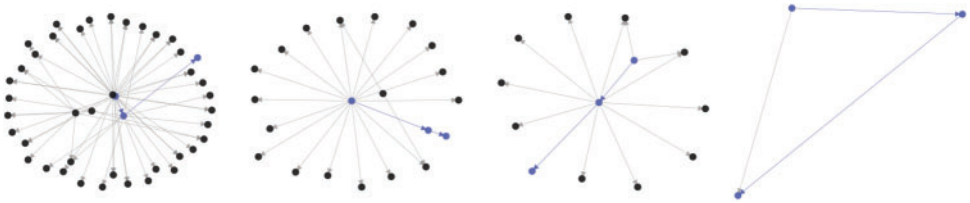

Figure 6: All subgraphs of CDG containing at least one DELEGATECALL sequence(marked in blue)

Algorithm 1: Calculating a dependency hash table for a subgraph of CDG

Input: A subgraph $SG = (V, E)$
Output: A dependency hash table H
1: $TS \leftarrow TopologicalSort(SG)$
2: **for** $k \in 0 \ldots TS.len$ **do**

(Continued)

Algorithm 1: (Continued)

```
3:    i ← InDeg(SG, TS[k])
4:    if i = 0 then
5:       H.insert(TS[k], ∅)
6:    else
7:       P ← Predecessors(SG, TS[k])
8:       T ← ∅
9:       for j ∈ 0 ... P.len do
10:         T ← T ∪ H.find(P[j]) ∪ {P[j]}
11:      H.insert(TS[k], T)
12: return H
```

4.3 Security Violation Analysis

Based on the definition of security violations against CDG, some subgraphs of CDG are marked if there is at least one vulnerable node in them. The automated vulnerability analysis of each node in a subgraph is implemented on the top of ETHBMC [17], a bounded model checker based on symbolic execution that supports the detection of vulnerabilities in a single contract, including the three types of known vulnerabilities discussed in Section 3.2. Moreover, other nodes that depend on the vulnerable node in marked subgraphs are also exposed to risk, denoted as potential victim contracts.

Algorithm 2 shows our security violation analysis algorithm for a subgraph of CDG. The input is a CDG's subgraph $SG = (V, C)$ which is generated in the construction stage, and a dependency hash table H which is calculated by Algorithm 1 for SG. The output is two vulnerability reports R_b and R_p, where R_b contains the vulnerable contracts with any of the three security violations and R_p contains the potential victim contracts affected by these vulnerable contracts in SG.

To identify all potential victim contracts in a subgraph of CDG, we first conduct security violation analysis (line 6) based on the out-degree and in-degree of each node, where we call the ETHBMC to generate W, a set collecting all known vulnerabilities of the current contract considering that a contract may have more than one vulnerability. For each security violation in W, the currently analyzed contract x is marked as a vulnerable contract, followed by all contracts that store in $H.find(x)$ marked as the potential victims (lines 10–12). According to the definition of Betrayal, x is also marked as a potential victim contract if $Type(w)$ gets the type of known vulnerability as Steal-Ether (lines 15–16). Different from most existing works, whose vulnerability reports only serve a single contract, ours covers multiple contracts with inter contract dependencies recorded in Ethereum. Therefore, reports for all types of security violations in CDG can be used to generate smart contract blacklists for Ethereum after considering appropriate classification and updating schemes.

Algorithm 2: Our security violation analysis algorithm for a subgraph of CDG

Input: $SG = (V, E)$, $v = (A, C) \in V$
A dependency hash table H
Output: A vulnerability report R_b,
A potential victim report R_p

```
1:  R_b ← ∅
2:  R_p ← ∅
```

(Continued)

Algorithm 2: (Continued)

```
for all x ∈ V do
    A_x, C ← x
    i, o ← Degree(SG, x)
    W ← Violations(i, o, C)
    if ||W|| > 0 then
        for all w ∈ W do
            R_b ← R_b ∪ {(A_x, w)}
            Q ← H.find(x)
            if ||Q|| > 0 then
            for all y ∈ Q do
                A_y ← y
                R_p ← R_p ∪ {(A_x, A_y, w)}
            if Type(w) = StealEther then
                R_p ← R_p ∪ {(A_x, A_x, w)}
return R_b, R_p
```

5 Ethereum Smart Contract Vulnerabilities Landscape

We ran the above large-scale analysis on a Ubuntu machine with AMD Ryzen 5800X CPU (4.5 GHz) (8 cores and 16 threads in total) and 32 GB of memory. Additionally, we ran 16 ETHBMC instances on the machine, each of which uses Yices2 [31] as backend SMT solver with three minute timeout. And the analysis timeout of a single contract is limited to 30 min. Scanning 2,616 subgraphs with a total of 13,682 nodes took the machine around 1 month in total.

5.1 Results

Our analysis finished successfully for the majority of bytecode-unique contracts (73.10%), with a certain number of timeouts (26.89%) and a negligible number of contracts encountering errors during analysis (0.01%). Noting that the failure rate in our analysis is significantly higher than one done in ETHBMC [17] (8.79%), one of the possible reasons is that ETHBMC experiment additionally runs a modified Parity archive node [32] to restore the storage states for all contracts, but we adopt the same approach as the author of teEther [15] just initializing them to zero, which affects the code coverage of the analysis process (only 38.02% in average in our analysis). However, it is not the problem of our CDG-based large-scale analysis, but can be resolved by running a modified Parity archive node until the 13,000,000th block is synchronized, which means several TBs of hard drive and months of time. We may be able to do that in future work. Bytecode-unique contracts that are not successfully analyzed, including timeouts and errors, are marked as bug-free despite possible false negatives among them, but they can be marked as potential victim contracts once the corresponding security violation is detected.

5.2 Betrayal Subgraph

Our analysis has flagged 16 subgraphs of CDG that may have the security violation of Betrayal. Each subgraph has at least one vulnerable node (11 at most in our work) that is exposed to the Steal-Ether vulnerability. According to the definition of Betrayal, a total number of 31 vulnerable nodes make 62 nodes become potential victims, including vulnerable nodes themselves. On the one hand, the bug code on the callee contract could leak not only its own but also its callers' Ether to the attacker. On the other hand, a caller contract can act as an entry point for the attacker to successfully exploit

the callee's vulnerability by taking advantage of one of DELEGATECALL's features that retains the execution environment data of the original call. As shown in Fig. 7, all vulnerable contracts are callee contracts in our large-scale analysis, which supports the above two situations. Besides, since a potential victim contract has a large number of copies in Ethereum ecosystem, we find an additional 3,480 contract accounts that have exactly the same bytecode as potential victim contracts, all of them share the same risk.

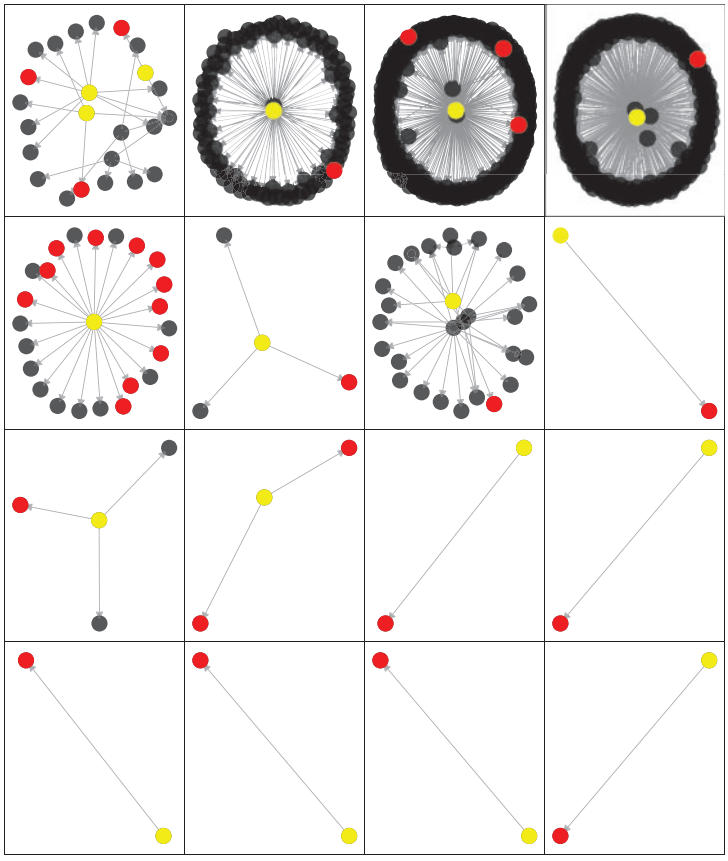

Figure 7: Topology of Betrayal subgraphs and distribution of different marked contracts

For any Betrayed subgraph, vulnerable contracts need to be replaced with bug-free contracts due to their immutability, while the potential victim contracts do not need to change their bytecodes, but change their persistent storages to other dependable smart contracts.

5.3 Deviation Subgraph

Six subgraphs of CDG are flagged with the security violation of Deviation, each of which has at least one vulnerable node suffering from a Code-Injection vulnerability. We find that six vulnerable nodes make other three nodes become potential victims, since some vulnerable nodes in our large-scale analysis are caller contracts on which no other contracts depend, as shown in Fig. 8. Besides, an additional 25,235 contract accounts have exactly the same bytecode as potential victim contracts, and one contract account considered vulnerable. An attacker can hijack the control flow of the vulnerable contract to execute a well-designed shellcode by constraining the target address of the CALLCODE/DELEGATECALL to be an attacker-controlled contract account address [15,17]. In

other words, the attacker can add nodes to a Deviation subgraph without being restricted by access control, and thereby intrude a subset of nodes in the subgraph by making the vulnerable contract depend on these nodes.

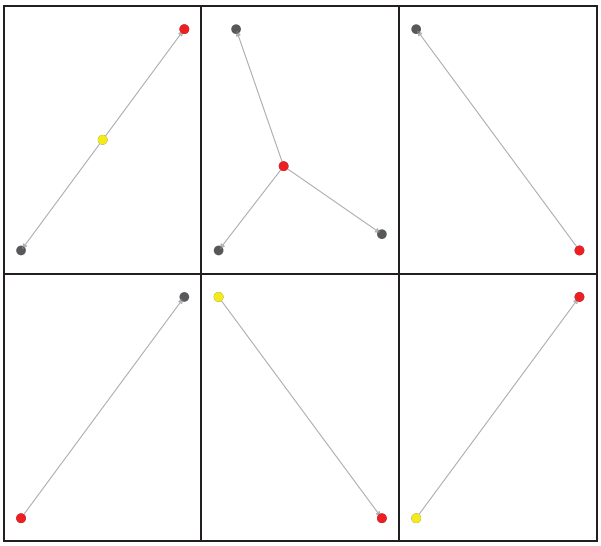

Figure 8: Topology of Deviation subgraphs and distribution of different marked contracts

For any Deviation subgraph, the vulnerable contracts in it need to be replaced by newly deployed contracts because they may be used for malicious code injection, and all potential victim contracts they bring either become re-dependent on the new contracts or are discarded.

5.4 Interruption Subgraph

Our analysis only flags one two-node subgraph of CDG with the security violation of Interruption. Since the only vulnerable node[1] having Suicidal vulnerability is a caller contract, there is no potential victim contract in such an interruption subgraph. Compared with the existing large-scale analysis results (see Table 2), there are two possible reasons for the significant decline in the number of Suicidal contracts. First, we only analyze contracts that are active (called at least 10 times) in the latest blocks. Second, this type of vulnerability has already been well contained throughout the Ethereum ecosystem, such as increasing the gas cost of SELFDESTRUCT [33].

Table 2: A survey of the evolution of the Ethereum smart contract vulnerabilities landscape

	teEther [15]	ETHBMC [17]	Our work
Time range	July 30 2015–Nov. 30 2017	July 30 2015–Dec. 24 2018	Jan. 2 2020–Aug. 10 2021
Block number range	genesis–around 4,646,800	genesis–around 6,944,150	9,200,000–13,000,000
Total contrat	38,757	2,194,650	195,247

(Continued)

[1] https://etherscan.io/address/0x52f93c5d9f23cc4ff86ada600d0d36bee86d99e4.

Table 2 (continued)

	teEther [15]	ETHBMC [17]	Our work
Steal-Ether contract	547 (1.4%)	2,708 (0.1%)	31 (0.02%)
Code-Injection contract	10 (<0.01%)	97 (<0.01%)	7 (<0.01%)
Suicidal contract	298 (0.77%)	1,924 (0.09%)	1 (<0.01%)
potential victim contract	-	-	28,786 (14.7%)

For any Interruption subgraph, before the vulnerable contracts are self-destructed, their Ether, as well as that of all potential victimized contracts should be transferred to avoid Locked Ether attacks.

5.5 Case Studies

In an effort to show these security violations in more detail, we manually reviewed all the instances of them and took those instances with Solidity source code as case studies in this section. Finally, we do not publish addresses of vulnerable contracts and potential victim contracts except for these cases to avoid malicious exploits.

TokenBuildInGenesis Bug. This instance corresponds to the subgraph(3,0) (from left to right, top to bottom) in Fig. 7. The only vulnerable contract marked in red is TokenBuildInGenesis contract, which can cause a Steal-Ether vulnerability by a sequence consisting of two transactions. The same vulnerability can be further repeated in the contract marked in yellow that depend on it with the help of DELEGATECALL to the same buggy code. Details about the contract code and the input data of several transactions are given in Appendix A.

AuthenticatedProxy Bug. According to the subgraph(0,0) in Fig. 8, the analysis report shows that the vulnerable contract, AuthenticatedProxy, has an erroneous visibility for its important initialization functions which may lead to its owner being changed without permission, resulting in the loss of necessary access controls when making proxy requests. The unexpected behavior can further spread to the only contract that relies on it. However, the unmarked contract in the subgraph will not be affected in any way, due to lack of similar dependencies. The relationship between them and the sequence of transactions that trigger the vulnerability are given in Appendix B.

DeleteContract Bug. There is only one instance of the Interruption Subgraph, which has one vulnerable contract without Solidity source code in etherscan.io, but does not have any potential victim contract. We manually reviewed on-chain data of the vulnerable one and found it is a library contract with a balance of 0. An attacker can exploit its Suicidal vulnerability through a sequence of two transactions, where the first transaction calls the function a91ee0dc to replace the owner of the smart contract, the second transaction calls the function 41c0e1b5 containing the SELFDESTRUCT instruction, causing the contract to self-destruct and transfer the balance to new "owner". We do not give the complete input data of two transactions, considering this contract has not been self-destructed as for now.

5.6 Current Vulnerability Landscape

We collected the results of large-scale analyses on Ethereum contracts in recent years, and presented a comparison of relevant data in Table 2. We note that ETHBMC runs an additional Ethereum archive node to verify the vulnerabilities' exploitability and obtain the storage states of contracts it analyzed. Therefore, the data it provided is more convincing. Although our large-scale

analysis is based on the ETHBMC tool, we took the same approach as teEther to initialize the storage state of the contract with an empty value, which may lead to false positives or false negatives to some extent, instead of relying on an Ethereum archive node. Different from previous works, we took the contracts active in the latest block as targets, marked the potential victim contracts other than the vulnerable ones, and found that they occupy a non-negligible proportion.

Furthermore, considering the time evolution of several large-scale analyses in Table 2, we can draw the following four conclusions about the current Ethereum smart contract vulnerabilities landscape. First, vulnerable contracts do not account for a high proportion of the entire Ethereum contract ecosystem. Second, the impact of different types of vulnerabilities on the latest blocks of Ethereum has changed, and contracts in earlier blocks have more vulnerabilities than those in the latest blocks. Third, the absence of vulnerabilities in the bytecode of a contract itself does not mean that it is protected from attacks, more contracts may be warned if considering other contracts on which it depends in Ethereum. Fourth, contract homogeneity is still obvious in the latest block (a contract is repeated 25,232 times in our analysis, see Appendix B), and to some degree can extend the impact of one vulnerability.

6 Conclusion

An increasing number of complex decentralized applications require more than one contract to work together, and these contracts often use DELEGATECALL-type internal transactions to communicate, which make the caller contract depend on the callee contract at a bytecode or a storage level. However, little is known about how these inter-contract dependencies affect smart contract vulnerability analysis, specifically, which contracts are affected and how the inter-contract dependencies affect them. This paper presents a vulnerability analysis scheme for Ethereum smart contracts based on inter-contract dependencies is presented to solve these problems. Specifically, we first present the definition of contract dependency graph (CDG) to capture inter-contract dependencies in Ethereum, and then define three categories of security violations against CDG, namely Deviation, Interruption and Betrayal, based on the location and characteristics of different vulnerable contracts. To identify both vulnerable and potential victim contracts that could be affected by the above three security violations, a large-scale analysis that focuses on three specific known vulnerabilities is designed and implemented on the latest Ethereum. Our large-scale analysis successfully finds 38 vulnerable contracts along with 28,786 potential victim contracts in total when considering the Ethereum smart contract homogeneity. These potential victims either suffer from out of service, are injected by malicious contracts, or invoke bug codes in their own context. Our scheme can be extended to more types of known vulnerabilities than the three specific known vulnerabilities above and further used for generating smart contract blacklists for Ethereum. In contrast, the exploitability of marked potential victim contracts and management of blacklists remains to be explored in the future.

Funding Statement: This work was supported by the Key R and D Programs of Zhejiang Province under Grant No. 2022C01018 and the Natural Science Foundation of Zhejiang Province under Grant No. LQ20F020019.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

1. Wang, Z., Jin, H., Dai, W., Choo, K. K. R., Zou, D. (2021). Ethereum smart contract security research: Survey and future research opportunities. *Frontiers of Computer Science, 15(2),* 1–18. DOI 10.1007/s11704-020-9284-9.
2. Vogelsteller, F., Buterin, V. (2015). Eip-20: Token standard. https://eips.ethereum.org/EIPS/eip-20.
3. Entriken, W., Shirley, D., E., J., Sachs, N. (2018). Eip-721: Non-fungible token standard. https://eips.ethereum.org/EIPS/eip-721.
4. Radomski, W., Cooke, A., Castonguay, P., Therien, J., Binet, E. (2018). Eip-1155: Multi token standard. https://eips.ethereum.org/EIPS/eip-1155.
5. Hasan, H. R., Salah, K. (2018). Proof of delivery of digital assets using blockchain and smart contracts. *IEEE Access, 6,* 65439–65448. DOI 10.1109/ACCESS.2018.2876971.
6. Luu, L., Chu, D. H., Olickel, H., Saxena, P., Hobor, A. (2016). Making smart contracts smarter. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 254–269. Vienna.
7. Day, A., Medvedev, E. (2018). Ethereum in bigquery: A public dataset for smart contract analytics. https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics.
8. He, N., Wu, L., Wang, H., Guo, Y., Jiang, X. (2020). Characterizing code clones in the ethereum smart contract ecosystem. *International Conference on Financial Cryptography and Data Security*, pp. 654–675. Malaysia.
9. Kiffer, L., Levin, D., Mislove, A. (2018). Analyzing ethereum's contract topology. *Proceedings of the Internet Measurement Conference 2018*, pp. 494–499. Boston.
10. Chen, T., Li, Z., Zhu, Y., Chen, J., Luo, X. et al. (2020). Understanding ethereum via graph analysis. *ACM Transactions on Internet Technology, 20(2),* 1–32. DOI 10.1145/3381036.
11. Parity, T. (2017). The multi-sig hack: A postmortem. https://www.parity.io/blog/the-multi-sig-hack-a-postmortem.
12. Parity, T. (2017). A postmortem on the parity multi-sig library self-destruct. https://www.io/blog/a-postmortem-on-the-parity-multi-sig-library-self-destruct/.
13. Parity, T. (2017). Security alert. https://www.parity.io/blog/security-alert-2/.
14. Etherscan (2017). Parity walletlibrary. https://etherscan.io/address/0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4#code.
15. Krupp, J., Rossow, C. (2018). Teether: gnawing at ethereum to automatically exploit smart contracts. *Proceedings of the 27th USENIX Security Symposium*, pp. 1317–1333. Baltimore.
16. Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A. (2018). Finding the greedy, prodigal, and suicidal contracts at scale. *Proceedings of the 34th Annual Computer Security Applications Conference*, pp. 653–663. San Juan.
17. Frank, J., Aschermann, C., Holz, T. (2020). {ETHBMC}: A bounded model checker for smart contracts. *Proceedings of the 29th USENIX Security Symposium*, pp. 2757–2774. Berkeley.
18. Chen, J., Xia, X., Lo, D., Grundy, J., Luo, X. et al. (2021). Defectchecker: Automated smart contract defect detection by analyzing evm bytecode. *IEEE Transactions on Software Engineering, 5589,* 1–19. DOI 10.1109/TSE.32.
19. Atzei, N., Bartoletti, M., Cimoli, T. (2017). A survey of attacks on ethereum smart contracts (SoK). *International Conference on Principles of Security and Trust*, Springer.
20. Saini, V. (2020). Writing upgradable smart contracts. https://hackernoon.com/smart-contract-versioning-mr5 dB x32db.
21. Murray, P., Welch, N., Messerman, J. (2018). Eip-1167: Minimal proxy contract. https://eips.ethereum.org/EIPS/eip-1167.

22. Yang, X., Liu, J., Li, X. (2020). Implementation smart contract with finite state machines. *Proceedings of the 2020 International Conference on Aviation Safety and Information Technology*, pp. 404–408. Weihai, China.
23. Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper, 151,* 1–32.
24. Ethervm (2021). Ethereum virtual machine opcodes. https://ethervm.io/#3A.
25. So, S., Hong, S., Oh, H. (2021). Smartest: Effectively hunting vulnerable transaction sequences in smart contracts through language model-guided symbolic execution. *Proceedings of the 30th USENIX Security Symposium*, pp. 1361–1378. Vancouver.
26. Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G. et al. (2019). Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1186–1189. San Diego.
27. Mueller, B. (2018). Smashing ethereum smart contracts for fun and real profit. *HITB SECCONF Amsterdam*, vol. 9, pp. 54.
28. jdourlens (2020). Using safe math library to prevent from overflows. https://ethereumdev.io/using-safe-math-library-to-prevent-from-overflows/.
29. NccGroup (2018). Decentralized application security project top 10 of 2018. https://eips.ethereum.org/EIPS/eip-1167.
30. Perez, D., Livshits, B. (2021). Smart contract vulnerabilities: vulnerable does not imply exploited. *Proceedings of the 30th USENIX Security Symposium*, pp. 1325–1341. Vancouver.
31. Dutertre, B. (2014). Yices 2.2. *International Conference on Computer Aided Verification*, pp. 737–744. Vienna, Austria.
32. Frank, J. (2020). Openethereum. https://github.com/Joool/openethereum.
33. Chen, T., Li, X., Wang, Y., Chen, J., Li, Z. et al. (2017). An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. *Information Security Practice and Experience (ISPEC)*, pp. 3–24. Melbourne, Australia.

Appendix A. TokenBuildInGenesis Bug

Code 1. TokenBuildInGenesis contract

```
contract TokenBuildInGenesis is DSAuth, SettingIds {
    ISettingsRegistry public registry;
    bool public paused = false;
    bool private singletonLock = false;
    modifier singletonLockCall () {
        require (!singletonLock, "Only_can_call_once");
        _;
        singletonLock = true;
    }
    function initializeContract (address _registry, bool _status)
        public singletonLockCall{
            registry = ISettingsRegistry (_registry);
            paused = _status;
```

```
        owner = msg . sender;
    }
    function claimTokens (address _token) public auth {
        if (_token == 0x0) {
            owner . transfer (address (this) . balance);
            return;
        }
        ERC20 token = ERC20 (_token);
        uint balance = token . balanceOf (address (this));
        token . transfer (owner, balance);
        emit ClaimedTokens (_token, owner, balance);
    }
}
```

When analyzing a Betrayal subgraph composed of two bytecode-unique contracts A(0x9620bfdce7184d246bf793207d27a99f75981c1e[2]), and B(0x7550aab5dfda37f646b1a826f27e9d604d8d1579[3]).

a Steal-Ether vulnerability is detected in contract A, and EthBMC successfully generated the an exploit consisting of two transactions in 78 s (see Transactions 1, 2).

Transaction 1

data:	8b3a f1f1 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 8000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 8000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 8000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 8000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Transaction 2

data:	df8d e3e7 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 8000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 8000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 8000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 8000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

[2] https://etherscan.io/address/0x9620bfdce7184d246bf793207d27a99f75981c1e.
[3] https://etherscan.io/address/0x7550aab5dfda37f646b1a826f27e9d604d8d1579.

Code 1 above is taken verbatim from part of the verified original source code of TokenBuildInGenesis contract(i.e., contract A). Combined with the ABI of it, we briefly explain the process of exploiting the vulnerability: the first transaction of this exploit calls function *initializeContract(address,bool)* (8b3af1f1) to set attacker-controlled address("0000000000000000000000000000000 0080000000" in this case) as owner, then the second transaction calls *claimTokens(address)* (df8de3e7) which transfer all balance of contract A to its owner when parameter is set to 0.

Therefore, contract A is marked as a vulnerable contract, and according to the definition of the Betrayal-type security violation, while the only contract B that depends on it is marked as a potential victim one. By DELEGATECALL to the *claimTokens(address)* in contract A after a simple transformation of Transaction 2, the same vulnerability will be repeated on contract B.

Appendix B. AuthenticatedProxy Bug

Code 2. AuthenticatedProxy contract

```
contract AuthenticatedProxy is TokenRecipient, OwnedUpgradeabilityStorage {
    bool initialized = false;
    address public user;
    ProxyRegistry public registry;
    function initialize (address addrUser, ProxyRegistry addrRegistry) public{
        require (!initialized);
        initialized = true;
        user = addrUser;
        registry = addrRegistry;
    }
    function proxy (address dest, HowToCall howToCall, bytes calldata)
        public returns (bool result) {
        require (msg . sender == user ||
           (!revoked && registry . contracts (msg . sender))
        );
        if (howToCall == HowToCall.Call) {
           result = dest . call (calldata);
        } else if (howToCall == HowToCall . DelegateCall) {
           result = dest . delegatecall (calldata);
        }
        return result;
    }
}
```

When analyzing a Deviation subgraph composed of three bytecode-unique contracts A (0x70e1e07d64 d4c0476ff45382a21a0b0205885489[4]), B (0xc99f70bfd82fb7c8f8191fdfbfb735606b15e5 c5[5]), and C (0xf9 e266af4bca5890e2781812cc6a6e89495a79f2[6]), whose adjacency matrix is as follows:

$$\begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

a Code-Injection vulnerability is detected in contract C, and EthBMC successfully generated the an exploit consisting of two transactions in 147 s (see Transactions 3, 4).

Transaction 3

data:	485c c955 0000 0000 0000 0000 0000 0000 0dfa 72de 72f9 6cf5 b127 b070 e90d 68ec 9710 797c 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 8000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 8000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 8000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Transaction 4

data:	1b0f 7ba9 0000 0000 0000 0000 0000 0000 f9c3 1051 1569 5a35 c255 88d4 e768 c6c2 e573 338d 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001 8000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 8000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 8000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Code 2 above is taken verbatim from part of the verified original source code of AuthenticatedProxy contract (i.e., contract C). Combined with the ABI of it, we briefly explain the process of exploiting the vulnerability: the first transaction of this exploit calls function *initialize(address,address)* (485cc955) to re-initialize the contract's owner information, then the second transaction calls *proxy(address,uint8,bytes)* (1b0f7ba9) which sends DELEGATECALL-type message call to user-control destination address(0xf9c . . . 38d).

Therefore, contract C is marked as a vulnerable contract, and according to the definition of the Deviation-type security violation, contract A is marked as a potential victim one. Besides, there are at least 25,232 contracts with the same bytecode as contract A, whose address can be viewed in the internal transaction history[7] of contract C.

[4] https://etherscan.io/address/0x70e1e07d64d4c0476ff45382a21a0b0205885489.
[5] https://etherscan.io/address/0xc99f70bfd82fb7c8f8191fdfbfb735606b15e5c5.
[6] https://etherscan.io/address/0xf9e266af4bca5890e2781812cc6a6e89495a79f2.
[7] https://etherscan.io/txsinternal?ps=100&zero=false&a=0xf9e266af4bca5890e2781812cc6a6e89495a79f2&valid=all&m=advanced.