Diving Into Blockchain’s Weaknesses: An Empirical Study of Blockchain System Vulnerabilities

XIAO YI, Department of Information Engineering
The Chinese University of Hong Kong
DAOYUAN WU*, Department of Information Engineering
The Chinese University of Hong Kong
LINGXIAO JIANG, School of Computing and Information Systems
Singapore Management University
KEHUAN ZHANG, Department of Information Engineering
The Chinese University of Hong Kong
WEI ZHANG, School of Computer Science
Nanjing University of Posts and Telecommunications

Blockchain is an emerging technology for its decentralization and the capability of enabling cryptocurrencies and smart contracts. However, as a distributed ledger software by nature, blockchain inevitably has software issues. While application-level smart contracts have been extensively investigated, the underlying system-level security bugs of blockchain are much less explored. In this paper, we conduct an empirical study of blockchain’s system vulnerabilities using four representative blockchains, Bitcoin, Ethereum, Monero, and Stellar. Due to the lack of CVE information associated with these blockchain projects, we first design a systematic process to effectively identify 1,037 vulnerabilities and their 2,317 patches from 34,245 issues/PRs (pull requests) and 85,164 commits on GitHub. This allows us to build the first blockchain vulnerability dataset, which will be released to the community as a part of our contributions. Atop this unique dataset, we perform three levels of analyses, including (i) file-level vulnerable module categorization by identifying and correlating module paths across projects, (ii) text-level vulnerability type clustering by combining natural language processing with similarity-based sentence clustering, and (iii) code-level vulnerability pattern analysis by generating and clustering the code change signatures that concisely capture both syntactic and semantic information of patch code fragments.

Among detailed results, our analysis reveals three key findings, including (i) some blockchain modules are more susceptible than the others; notably, the modules related to consensus, wallet, and networking are highly susceptible, each with over 200 issues; (ii) around 70% of blockchain vulnerabilities are in traditional types, but we also identify four new types specific to blockchains; and (iii) we obtain 21 blockchain-specific vulnerability patterns that check unique blockchain attributes and validate various blockchain statuses, and demonstrate that they can be used to detect similar vulnerabilities in other top blockchains (e.g., Dogecoin and Bitcoin SV).

1 INTRODUCTION

While blockchain was first invented as a transaction ledger of the Bitcoin cryptocurrency [45], it is now serving as a fundamental component of many cryptocurrencies, the total market capitalization of which has surpassed two trillion USD in early April 2021 [32]. Smart contract platforms (e.g., Ethereum [14] and Hyperledger Fabric [9]) and decentralized computing platforms (e.g., Interplanetary File System [13] and Blockstack [8]) further evolved the blockchain technology into various decentralized applications, such as DeFi (Decentralized Finance) [55], smart contract oracles [61, 62], decentralized identities [43], decentralized IoT management [49], and decentralized app markets [17]. To protect the decentralization of these systems and secure those finance-critical cryptocurrencies, security is a top priority of many blockchains.

*Corresponding author. Contact: dywu@ie.cuhk.edu.hk.
Existing research on blockchain security mainly focused on smart contract vulnerability detection and network security analysis. Specifically, many static program analysis tools, e.g., Oyente [42], Zeus [29], Securify [52], Gigahorse [25], and ETHBMC [21], have been proposed to detect vulnerable smart contracts via symbolic execution and model checking. Dynamic tools [19, 27, 47, 51] and learning-based tools [24, 38, 41] were also invented. Besides application-level contract analysis, some works analyzed network-level hijacking [10] and mining [23] attacks and performed transaction-level attack analysis [20, 31, 63, 66]. In contrast, system-level security issues of the blockchain itself are much less explored in academic research. To the best of our knowledge, there was only one study [53] (from the software engineering community) in this direction. It specifically analyzed 946 blockchain bugs, with only 18 security bugs covered and four analyzed.

In this paper, we aim to systematically understand blockchain system vulnerabilities by conducting an empirical vulnerability study of the representative blockchains in four directions, including the classic Bitcoin [45], the smart contract platform Ethereum [14], the anonymous coin Monero [46], and the payment network Stellar [39]. They are not only popular in the cryptocurrency market but also backed up with solid technical papers.

As depicted in Figure 1, the first step and challenge of our study is to effectively collect vulnerable issues and their patches of those four blockchains. This is difficult because there is very little CVE information associated with blockchain projects (unlike other vulnerability mining studies [35, 56, 65]), and the large number (over 34K) of raw blockchain bugs in our crawled database makes manual vulnerability filtering 1 ineffective. To address this, we propose a vulnerability filtering framework based on the intuition that vulnerabilities have unique characteristics at different levels of bug attributes, and we can gradually identify candidate vulnerabilities by analyzing bug attributes from coarse-grained to fine-grained levels. Specifically, we perform the filtering at the commit, file, label, and keyword levels. Eventually, we obtain 1,037 vulnerabilities and their 2,317 patches as our blockchain vulnerability dataset.

Based on this unique dataset, we study three key yet unexplored aspects of blockchain vulnerabilities, including susceptible blockchain modules, common blockchain vulnerability types, and blockchain-specific patch code patterns. To this end, we perform the file-, text-, and code-level vulnerability analysis as follows.

Firstly, we conduct the module analysis by inspecting patched files. However, inspecting each individual file is time-consuming because there are 2,362 unique patch file paths. Therefore, we propose to identify the module path, i.e., the folder name that could summarize the module of enclosed files (e.g., the “rpc/” folder indicates the RPC module). We further correlate module paths across different blockchains by identifying a reference blockchain architecture and mapping different module paths into this architecture. The file-level module categorization allows us to obtain a layered map of blockchain vulnerabilities in different modules and pinpoint susceptible blockchain modules. We find that some modules are more susceptible than the others, such as the highly susceptible ones related to consensus, wallet, and networking (each with over 200

1That said, we need to recognize or differentiate real vulnerabilities from regular bugs.
issues). Besides that they are commonly used in blockchain systems, we show that their patch code complexity is higher than that in non-susceptible modules. Moreover, we further analyze and compare the vulnerability module differences across different blockchains.

Secondly, we perform the type analysis by analyzing vulnerability text, more specifically, vulnerability titles. This is because a vulnerability type is typically captured by the title of an issue/PR (pull request), e.g., Bitcoin PR #17640 “wallet: Fix uninitialized read in bumpfee(...)”, where “uninitialized read” is the type. To eliminate noisy words and generate good-quality clusters about types, we leverage the part-of-speech analysis of NLP (natural language processing) to first extract type keywords before we conduct actual clustering, based on the grammatical pattern that a type is often located in between a verb (e.g., “fix”) and a preposition (e.g., “in”). By extracting type keywords in various situations and identifying a suitable clustering algorithm (and its setting), we successfully map 75.8% of the vulnerabilities into the clusters of different types and analyze the top 20 types that affect at least ten vulnerabilities each. Four new types are specific and three are partially specific to blockchains, whereas traditional vulnerability types still hold 62%–78% of all the vulnerabilities.

Thirdly, we conduct the pattern analysis by analyzing vulnerability patch code. In particular, we focus on blockchain-specific vulnerability types since the code patterns of traditional vulnerability types are well-known. To facilitate similar patch code into the same cluster, we design and generate the code change signatures that concisely capture both syntactic and semantic information of patch code fragments. By clustering 3,251 code fragments into 174 clusters of code change signatures, we identify 21 blockchain-specific vulnerability patterns that check unique blockchain attributes (e.g., the sender address, transaction order, block header, and gas limit) and validate various blockchain statuses during node synchronization, peer validation, wallet, and database operations. We further leverage these patterns to discover 23 similar vulnerabilities in other top blockchains. The affected six blockchain projects include the well-known Dogecoin, Bitcoin SV, and Zcash, with a collective market capitalization of over 40 billion USD as of October 2021. Most of our vulnerability reports have been confirmed by their developers, with patches being underway. This demonstrates the real-world impact of our obtained vulnerability patterns.

To sum up, this paper provides a set of methodologies to analyze blockchain vulnerabilities, builds a knowledge base, and digs out the insights about them. To facilitate future research, we will release the collected dataset to the community.

2 BACKGROUND

In this section, we introduce the background of four representative blockchains we study in this paper and the typical bug-fixing process in these blockchain projects.

2.1 Four Representative Blockchains Studied

In this paper, we study the representative blockchains that are (i) popular in the cryptocurrency market, (ii) in different directions of blockchain usages, and (iii) backed up with solid technical papers. Under these three conditions, we select the classic Bitcoin [45], the smart contract platform Ethereum [14], the anonymous coin Monero [46], and the payment network Stellar [39]. Next, we present their basic information and the current development status on GitHub.

**Bitcoin** is the first decentralized cryptocurrency and often described as “digital gold.” Bitcoin introduces the concept of blockchain [45] and uses it as a distributed ledger to record transactions for public verification. As of 18 October 2021, the Bitcoin cryptocurrency (or BTC) has the top one market capitalization of more than 1.17 trillion USD. The Bitcoin software was released in 2009, and it is now actively maintained by more than 820 contributors on GitHub in a repository called bitcoin/bitcoin. The primary programming language of Bitcoin is C++.
Ethereum is the first blockchain system with the capability of constructing Turing-complete smart contracts [14], which contain a set of pre-defined rules and regulations for self-executing. To maintain the operation of Ethereum, it creates a native cryptocurrency called Ether (or ETH), which is the second largest cryptocurrency with a market capitalization of more than 448 billion USD as of 18 October 2021. The Ethereum software was released on GitHub in 2015, and its Go implementation is maintained by 660 contributors in a repository called ethereum/go-ethereum.

Monero aims to mitigate the privacy leakage in blockchain systems, since each blockchain transaction is transparent and could leak some sensitive information. To do so, Monero uses an obfuscated ledger [46] to prevent the transaction details (e.g., transaction source, amount, and destination) from being revealed to outside observers. As of 18 October 2021, the Monero coin (XMR) is ranked 39th with a market capitalization of over 4.6 billion USD. The Monero software was released on GitHub in 2014, and it is maintained by more than 250 contributors in a repository called monero-project/monero. The primary language of Monero is C++.

Stellar is a blockchain-based payment network [39] that can perform cross-border money transfer in seconds. It uses a novel consensus protocol called Stellar Consensus Protocol (SCP) [39] for fast and secure transactions among untrusted participants. The native cryptocurrency of Stellar is called XLM, which is ranked 22nd with a market capitalization of around nine billion USD as of 18 October 2021. The Stellar software was released on GitHub in 2015, and it is currently maintained by more than 70 contributors in a repository called stellar/stellar-core. Similar to Bitcoin and Monero, the primary language of Stellar is also C++.

2.2 Bug-Fixing Process in Blockchain Projects

It is also necessary to understand the typical bug-fixing process of blockchain projects hosted as open-source projects on GitHub in order to collect and analyze their vulnerabilities and patches. There are three concepts, commits, issues, and pull requests. A commit is a set of changes submitted by developers into a project repository; the changes can be anything, ranging from changing code to modifying document files or merging multiple previous commits. An issue is often a report on a project’s GitHub page; it may describe a potential bug or sometimes an enhancement or a question, and may come with fixes and solutions. A pull request (PR) is the proposed commit for a project from a separate clone of the project; it can be pulled from the project clone and accepted into the original project based on the review of managing developers.

Usually, when a contributor identifies a bug, he/she will create an issue on the project’s GitHub page to report this bug. For example, Figure 2 shows that a contributor named “TheBlueMatt”...
identified a race in Bitcoin and reported this bug in the issue #11106 dated 21 Aug 2017. Later, a developer named “meshcollider” added four commits to fix this issue in the PR #11107. These commits in the PR are finally merged into the main repository by ‘MarcoFalke’ on 5 Oct 2017. In many other cases, a project developer could directly make a PR to fix code without creating an issue (actually 88% of the vulnerabilities in our dataset are directed patched via PRs). As a result, for simplicity, we do not explicitly distinguish an issue and a PR in this paper since the latter often contains a bug description too. Indeed, GitHub itself mixes up the usage of issue/PR numbers.

3 SYSTEMATIC DATA COLLECTION

As shown in Figure 1, the first and a critical step of our study is to collect a good-quality blockchain vulnerability dataset across multiple blockchain systems that satisfies two conditions: (i) cover as many as vulnerabilities in the studied blockchains (i.e., minimizing false negatives); and (ii) introduce as few as non-vulnerability bugs in the dataset (i.e., minimizing false positives).

There can be various ways to collect vulnerability data from project outsiders’ perspectives. One is to leverage the CVE (Common Vulnerabilities and Exposures) or Bulletin (i.e., bug bounty) information in a way similar to some other vulnerability studies [35, 56, 65]. However, we found that there is very little CVE/Bulletin information about most blockchains because blockchain vulnerabilities are critical and often patched directly via the reports from bug bounty programs without releasing a CVE. For example, Ethereum (go-ethereum) had only four CVEs released before our data collection while Bitcoin had 33 CVEs, which are significantly fewer than those we collect in this paper. Therefore, we take a second way to analyze blockchain projects’ issues and commits directly and extract the vulnerable ones from them. Specifically, we choose to analyze those blockchains’ GitHub repositories since they often provide detailed bug and vulnerability descriptions in their issues or PRs and patch code in the commits related to the issues or PRs (see §2.2).

To this end, we first crawl all blockchain bugs and organize them into a raw bug database (in §3.1). The major challenge is how to recognize or differentiate real vulnerabilities from a large number of regular bugs. We cannot leverage the existing training-based patch identification [50, 67] since (i) there is no ground-truth training set for blockchain vulnerabilities; and (ii) the learning-based nature tends to make them identify only the same classes of bugs or vulnerabilities. Moreover, the large number of blockchain bugs in our raw database, over 34K, makes manual analysis ineffective. Therefore, we propose a vulnerability filtering framework (in §3.2) that systematically and effectively filters out regular bugs and extracts blockchain vulnerabilities. By applying this novel method to the raw database, we eventually obtain the first dataset of blockchain system vulnerabilities (in §3.3), comprising 1K vulnerabilities from four major blockchains.

3.1 Crawling and Organizing Blockchain Bugs

As illustrated in Figure 1, our blockchain bug database is constructed from two data sources, the issues and commits. For the issues, we collect all the information of each closed issue/PR, including the issue title, issue body, comments, events, and bug category labels. We consider only closed issues/PRs because open issues are not confirmed bugs yet and certainly have no patches. Note that even for closed issues, they may not be the real bugs and could have no patches (i.e., they were simply closed by developers). For the commits, we first crawl all the commits of a repository and then determine which commits are bug-related. Specifically, we collect the title, commit message, affected files, and URL of each commit. For these raw data, we can leverage GitHub APIs to crawl

---

2Since the metadata of issues and the information for correlating a commit and an issue are available only on GitHub, we use the same crawling method to handle commits instead of interacting with cloned git repositories.
them directly without composing a web page crawler. We have collected a total of 34,245 closed issues/PRs and 85,164 commits at the end of February 2020. The detailed breakdown of these issues/PRs and commits across four blockchain projects is listed in the left part of Table 2.

With the raw data collected, a non-trivial task in this subsection is to organize and correlate the issues with their corresponding commits. Specifically, we need to determine all the relevant commits for a given issue/PR — if an issue/PR has no patch commits, it is not a real bug and will be filtered out. By summarizing the issue/PR and commit’s GitHub structures, we observe three kinds of information we can leverage for such correlation. First, we leverage the issue page’s event information (e.g., XXX mentioned this issue and YYY added a commit) and retrieve the commit URLs from those events. For example, in https://github.com/bitcoin/bitcoin/issues/595, we obtain the commit URL via the event of “laanwj added a commit that referenced this issue.” Second, for a PR like https://github.com/bitcoin/bitcoin/pull/9366, we can directly retrieve its commit lists at its “Commits” tab page. Although these two kinds of information is useful for most issues/PRs, some commits may not appear in the events of issues or commit lists of PRs. To overcome this, our script analyzes all the 85,164 commits’ titles and messages and identifies issue/PR numbers from them. With these strategies, we successfully build the relationship between the issues and commits and finish constructing the raw bug database shown in Figure 1.

However, the raw bug database did not yet contain patch code, which is required for code-level pattern analysis in §6. We are interested in the patch code for the bugs determined as vulnerabilities through our vulnerability filtering in §3.2. Specifically, there are a total of 1,059 vulnerable issues/PRs, which associate with 2,933 code commits. Our objective is to collect the complete (patch) code hunks of these 2,933 commits, including their added, removed, and neighboring context code lines for future use. We then develop a script to automatically parse code hunks from patch commits and save them in structured JSON formats for easy reference.

3.2 A Vulnerability Filtering Framework

To evolve the raw bug database into the final vulnerability dataset, we design a systematic vulnerability filtering framework expressed as a seven-step process (i.e., S0∼S4b in Table 1) to effectively differentiate vulnerabilities from regular bugs with limited manual work. The intuition is that vulnerabilities have unique characteristics at different levels of bug attributes, and we can gradually identify candidate vulnerabilities by analyzing bug attributes from coarse-grained to fine-grained levels. As shown in Table 1, we perform the filtering at the following four levels:

**Commit-based filtering.** Firstly, in the step S0, we leverage the most straightforward characteristic that a closed vulnerability must associate with code commits. In other words, an issue/PR without any commit could be excluded directly. Since we have already built the relationship between issues/PRs and commits in §3.1, we easily exclude 10,101 issues/PRs out of the entire 34,245 issues/PRs in the raw bug database.

**File-based filtering.** Secondly, we leverage two (patch) file-level characteristics to filter out the bugs that are certainly not vulnerabilities. The basic idea of these two characteristics is that the patch of a vulnerable issue/PR must make some real code changes, including changing files with actual source code and not containing only test code. Specifically, in the step S1, we determine the file types with actual source code (by their file suffixes) for four blockchains. An issue/PR whose commits do not modify any file in these types should be excluded. For example, there are 152 different file types for Bitcoin’s commits, but only these seven file types, [‘.cpp’, ‘.h’, ‘.py’, ‘.sh’, ‘.cc’, ‘.c’, ‘.java’], contain actual source code whereas other file types like ‘.yml’ and ‘.mk’ are unlikely related to vulnerabilities. This step filters out 3,798 more issues/PRs, with the remaining 20,346 further filtered by the step S2. Specifically, S2 excludes the test-only commits and their associated issues/PRs. With the file-based filtering, we exclude 22% (5,322/24,144) of the issues/PRs.
Label-based filtering. Thirdly, we leverage the label-level characteristic that certain bug labels could indicate whether an issue/PR is related to a vulnerability or not. For example, the ‘Privacy’ label marks privacy-related bugs in the Bitcoin project and the ‘obsolete:vuln’ label indicates the early-stage vulnerabilities of Ethereum. To avoid false positives, we are conservative to specify vulnerability labels — we assign only three labels (i.e., the ‘Privacy’, ‘obsolete:vuln’, and special label ‘SEC-XXX’ that appeared in the beginning of issue/PR titles) and mark their corresponding 56 issues/PRs explicitly as vulnerabilities in the step S3a. In contrast, there are much more labels clearly indicating non-vulnerability issues/PRs. Specifically, out of the entire 87 labels from four blockchain projects, we manually determine that 48 of them are not related to vulnerabilities, such as ‘Refactoring’, ‘Docs’, and ‘type:feature’. With these labels, we filter out their associated 4,400 issues/PRs in the step S3b. After this step, we have narrowed the filtering scope from 34,245 to 14,368 issues/PRs, a reduction of 58%.

Keyword-based filtering. Lastly, we directly check issues/PRs’ text based on the characteristic that some keywords could indicate an issue/PR vulnerable whereas others could imply an issue/PR not related to vulnerabilities. To this end, we first perform a word count analysis on the words in issue/PR titles and bodies, sort these words by their appearance frequency, and exclude the words that appear only once. We then group the words by their semantic similarity using the spaCy [4] NLP library. Since similar words are grouped together, we manually go through all the clusters to obtain a set of vulnerability-related words (Step S4a) or non-vulnerability words (Step S4b). Specifically, we obtain 62 clusters of vulnerability-related words and 79 clusters of non-vulnerability words, which allows us to automatically determine 1,227 vulnerable issues/PRs and exclude 6,330 irrelevant issues/PRs in the step S4a and S4b, respectively.

Eventually, our filtering framework extracted 1,283 (=1,227+56) suspicious issues/PRs (in the step S3a and S4a) from the entire 34,245 issues/PRs. We have manually examined all these candidates and confirmed that 1,059 of them were actually vulnerability-related. This suggests that our filtering achieves an accuracy of 82.5%. It handles 80.1% (27,434/34,245) of all the issues/PRs; the remaining 6,811 are discarded since they are after the step S4b and have a low chance to be vulnerabilities due to no relevant keywords.

### 3.3 The Vulnerability Dataset and Its Metadata

As mentioned in the end of §3.1, we then retrieve the code hunks for the identified 1,059 issues/PRs from their corresponding 2,933 commits. This allows us to further exclude 22 issues/PRs because they associate with “invalid” code commits through the code hunk analysis. Specifically, we identified 586 duplicate code commits whose code hunks were the same (e.g., https://github.com/bitcoin/bitcoin/commit/d4781ac6 and https://github.com/bitcoin/bitcoin/commit/8a445c56), for which we kept just one code commit for each duplicate pair. We also found 30 empty code commits where we were not able to obtain their code hunks due to disappeared (e.g., https://github.com/bitcoin/bitcoin/commit/7e193ff6) or large diffs (e.g., https://github.com/ethereum/go-ethereum/commit/34dde3e2). As a result, our final vulnerability dataset consists of 1,037 vulnerability-related issues/PRs and their 2,317 commits, as shown in Table 2. It is worth noting that while items in our dataset are all security patches, some of them are not conventionally technical vulnerabilities but more like...
security enhancements, such as upgrading weak crypto algorithms to strong ones. In this paper, we do not distinguish them.

In Table 2, we also list the metadata of each blockchain project. We can see that Bitcoin and Ethereum contribute 77.8% of the vulnerabilities in our dataset, whereas the percentages of Monero and Stellar vulnerabilities are relatively low. This is mainly because Bitcoin and Ethereum have much more code commits than the other two blockchains, holding a similar percentage (76.9%) of the entire 85,164 commits. Additionally, we notice that Stellar has around the same number of patches as Monero but the number of issues/PRs is three times lower (56 v.s. 178). The main reason is that Stellar developers tended to use one PR to cover multiple-bug fixes at the early stage of Stellar development.

Next, based on this unique dataset, we perform a comprehensive vulnerability analysis at three different levels in §4, §5, and §6, respectively.

4 FILE-LEVEL MODULE CATEGORIZATION

At the first-level of our study, we perform the module analysis by inspecting patched files. In this section, we first propose a lightweight method for categorizing vulnerable modules in §4.1, and then present the categorization result and its implication in §4.2.

4.1 Identifying and Correlating Module Paths for Vulnerable Module Categorization

To perform module categorization, a basic idea is to inspect the (patch) file names and their paths corresponding to those vulnerabilities. We found that 1,037 vulnerable issues/PRs (or more precisely, 2,317 patch commits) totally generated 2,362 unique file paths (544 in Bitcoin, 1,376 in Ethereum, 251 in Monero, and 191 in Stellar), which makes inspecting each individual file time-consuming. Therefore, we propose to identify the module path, i.e., the folder name that could summarize the module of enclosed files (e.g., the “rpc/” folder indicates the RPC module). For some paths of generic names (e.g., the “src/” folder), we need to consider its sub-folders as module paths. Since Ethereum’s folder structure is more complicated than the other three projects, we also consider three additional folders (the “core/”, “swarm/”, and “eth/” folders) to be generic, and consider their sub-folders as module paths. Eventually, we obtain a total of 146 module paths (28 in Bitcoin, 71 in Ethereum, 26 in Monero, and 21 in Stellar) from 2,317 patch commits in the four studied blockchains. This significantly reduces the workload of categorizing vulnerability modules.

However, since different blockchains have different path names for the same module (e.g., the Consensus module of Bitcoin/Ethereum is in “consensus/” while that of Stellar is in “src/scp/”), we need to further correlate those module paths across projects. Our solution is to identify a reference blockchain architecture and map different module paths into this architecture. Since many blockchains are based on Bitcoin, we use Bitcoin Core’s architecture [2] as our reference. For easier understanding, we separate the entire architecture into four layers [1], as shown in Figure 3a, and unify the traditional Miner, Mempool, and Validation Engine components into the Consensus module. We then manually map those 146 module paths into our blockchain architecture one by one — a detailed mapping between four blockchains’ module paths and their unified modules will be released together with our blockchain vulnerability dataset.

It is worth noting that a vulnerable issue/PR may affect multiple modules, so the sum of all modules’ vulnerability numbers will be larger than 1,037. Also, some patch commits change only the files under the generic “src/” folder and do not have module paths. We manually inspect all such patch files (107 in Bitcoin, 31 in Ethereum, 6 in Monero, and 4 in Stellar) and map their corresponding vulnerabilities into the modules in Figure 3a based on the patch file names.
4.2 Susceptible Blockchain Modules

Figure 3a shows the result of our module categorization in the form of a layered map of blockchain modules and the numbers of vulnerabilities in those modules. We can see that modules in the Policy, Peer, Network layers each introduce around one-fourth of the vulnerabilities, while the UI modules and other uncategorized modules contribute the remaining 30%. Among all modules, we find that some modules are more susceptible than the others. Notably, the modules related to Consensus, Wallet, and NetConn are highly susceptible, each with over 200 issues. Other modules about RPC, GUI/CMD, and Storage are also susceptible, affecting around 100 issues each. We now highlight these modules in a bottom-up order:

- The Consensus module covers the consensus (e.g., the Proof-of-Work mechanism [45]), miner, block/transaction related components. Unfortunately, it was affected by 265 vulnerabilities, with the major module path from the “consensus/” folder. Other module paths include “miner/”, “ethchain/”, “src/cryptonote_core/”, “src/scp/”, and “src/ledger/”.
- In the Peer layer, the Wallet module handles transactions for each peer and the Storage module manages the storage of those transactions. As shown in Figure 3a, the Wallet module was affected by 214 vulnerabilities, which are mainly from the “src/wallet/” and “accounts/” module paths. In contrast, the Storage was affected by 93 vulnerabilities, all of which are from database-related module paths, such as “src/blockchain_db/”, “src/leveldb/”, and “ethdb/”.
- The NetConn and RPC modules collectively incurred the most blockchain vulnerabilities in our dataset. As a distributed system by nature, blockchain systems heavily rely on network synchronization and RPC (Remote Procedure Call). Since it deals with complex network communication of different peers, multiple security issues could occur, such as data race, deadlock, resource leak, and denial-of-service.
- Surprisingly, the GUI/CMD module is also a major source of vulnerabilities, with a total of 141 vulnerabilities from the module paths like “src/qt/”, “ethereum/”, “src/daemon/”, and “cmd/”. The underlying faults vary, but segfault and deadlock are typical bugs.

Besides that these modules are commonly used in blockchain systems, we further investigate other possible factors that cause them more susceptible. One intuition is that their code is more complex. Therefore, we count the code lines required for patching a vulnerability in each module, with the median marked in Figure 3a. We can see that for the modules in the same layer, the ones with more vulnerabilities are typically with larger median of changed file lines. For example, the NetConn module has the largest median of 17 in the Network layer and the Wallet also
has the largest median of 9 in the Peer layer. The only exception is the Crypto module, which has fewer vulnerabilities but larger median than the Consensus module. This is mainly because vulnerability patches in the Crypto module often replaced or updated the crypto algorithm block, e.g., https://github.com/monero-project/monero/pull/1945/commits/5ceecd3f.

After understanding the generalized map of blockchain vulnerability modules, we further compare their differences across different blockchain projects. First, we analyze the number of affected modules per vulnerability issue/PR and draw the CDF (Cumulative Distribution Function) plot for all four blockchains, as shown in Figure 3b. We can see that for Bitcoin, Ethereum, and Monero, over 85% of their vulnerability issues/PRs affect only one or two modules. In contrast, Stellar tends to cover more affected modules per vulnerability issue/PR, with the CDF plot clearly behind that of the other three. This result is consistent with our conjecture in §3.3 that Stellar developers use one PR to cover multiple-bug fixes at the early stage of Stellar development. Second, we observe that the most susceptible module of Monero is notably different that in other major blockchains. For example, the most susceptible modules of Bitcoin and Ethereum is RPC (19.2%) and consensus (23.08%), respectively, whereas around 40% of the Monero vulnerabilities affect the wallet module. This is likely because Monero has a higher level of anonymity, which requires a much more complicated procedure for key management and privacy leakage prevention in the wallet module.

**Key Takeaway in §4.2:** We obtained a layered map of blockchain vulnerabilities in different modules, and showed that some blockchain modules are more susceptible than the others. Notably, the modules related to consensus, wallet, and networking are highly susceptible, each with over 200 issues. We further investigated the possible factors and compared the module differences across different blockchains.

5 TEXT-LEVEL TYPE CLUSTERING

At the second-level of our study, we conduct the type analysis by analyzing vulnerability text. In this section, we first present a NLP-based approach for clustering vulnerability types in §5.1, and then summarize the clustering results and showcase common blockchain vulnerability types in §5.2, including the ones not known before.

5.1 NLP-based Analysis of Vulnerability Titles for Type Clustering

We find that a vulnerability type is typically captured by the title of an issue/PR page, e.g., Bitcoin PR #17640 “wallet: Fix uninitialized read in bumpfee(...)” where “uninitialized read” is the type. However, simply clustering issue/PR titles does not generate good-quality clusters about vulnerability types because each title could have some noises. For instance, in the earlier example, “wallet” and “bumpfee” would affect the clustering quality. To address this problem, we propose a novel NLP-based method to first extract **type keywords** before we conduct actual clustering. This method is based on a grammatical pattern of vulnerability titles we observed, that a type is often a noun phrase located in between a verb (e.g., “fix”) and a preposition (e.g., “in”). Figure 4 shows an intuitive illustration. Overall, our approach consists of two major steps: NLP-based keyword extraction and clustering the obtained type keywords. Before these two steps, we also need to perform some pre-processing.

**Pre-processing.** Before applying the NLP analysis for keyword extraction, we perform some pre-processing of issue/PR titles so that they are cleaned for next-stage analysis. To this end, we remove useless words and formalize remaining words in the vulnerability titles. Specifically, the useless words include (i) the module/version information (e.g., the word before “:”, such as the “wallet” above, or the word inside “[“], such as “[rpc]” or “[RELEASE]”), (ii) the special word (e.g., “SEC-*” for Ethereum and one-character word like “a”; note that numbers and symbols like “–” or “(...)” could be
An Empirical Study of Blockchain System Vulnerabilities
Technical Report from IE, CUHK, October 2021

Table 3. Examples of the cleaned issue/PR titles and their corresponding type keywords extracted.

| ID | Raw Title                                                                 | Cleaned Title                                                                 | Type Keywords            |
|----|---------------------------------------------------------------------------|------------------------------------------------------------------------------|--------------------------|
| E1 | accounts: fix two races in the account manager                         | [fix, two, races, in, the, account, manager]                                | [two, races]             |
| E2 | blockchain_db: sanity check on tx/hash vector sizes                      | [sanity, check, on, transaction, hash, vector, sizes]                        | [sanity, check]          |
| E3 | [net] Avoid possibility of NULL pointer dereference                      | [avoid, null, pointer, dereference]                                         | [null, pointer, dereference] |
| E4 | wallet: Fix uninitialized read in bumpfee (...)                          | [fix, uninitialized, read, in, bumpfee]                                      | [initialized, read]      |
| E5 | Prevent DOS attacks on in-flight data structures                         | [prevent, dos, attacks, on, in, flight, data, structures]                    | [dos, attacks]           |

Fig. 4. An example title to illustrate the grammatical pattern of vulnerability titles we observed.

automatically handled by tokenizing), and (iii) noun-like adjective words (e.g., “possibility of” and “use of”). After cleaning useless words, we further formalize the remaining words by setting them to the lower case and tokenizing them via the NLP n1tk library’s RegexpTokenizer. During this process, we also unify a few words (e.g., replacing all “tx”/“txs”/“txns” using “transaction”). In Table 3, we list several example titles our script automatically cleaned.

NLP-based keyword extraction. According to the grammatical pattern shown in Figure 4, our objective is to find the target verb and preposition that could determine the range of type words. However, one vulnerability title may contain multiple verbs or prepositions. Moreover, some verbs mainly act as nouns in our context, such as “check” and “leak”. Based on these two reasons, we do not directly use the n1tk library’s pos_tag() for a real-time part-of-speech analysis. Instead, we perform a pre-analysis of words’ parts of speech in our cleaned vulnerability titles and build a vocabulary of verbs and prepositions and count their frequency in our dataset. Eventually, we obtain a list of 33 verbs and 21 prepositions and rank them by frequencies. Table 4 shows the top 10 frequently used verbs and prepositions in our dataset.

Table 4. The top 10 frequently used verbs and prepositions.

| Verb | fixed | remove | add | avoid | make | improve | handled | fixed | added |
|------|-------|--------|-----|-------|------|---------|---------|-------|-------|
|      |       |        |     |       |      |         |         |       |       |
| Preposition | in | for | on | of | with | from | by | before | if | after |

Based on our vocabulary of verbs and prepositions and their frequencies, we are able to automatically locate the target verb and preposition for a cleaned vulnerability title in various situations using the following rules:

- If only one verb and one preposition exist and the preposition appears after the verb (with one or more words in between), such a verb and preposition, e.g., the word fix and in of the example E1 in Table 3, are the target words.
- If there is no verb but the preposition exists (e.g., the example E2) or there is no preposition but the verb exists (e.g., the example E3), the preposition or the verb will be determined as the target, respectively.
- If multiple verbs appear in a title, the one with the highest frequency will be regarded as the target verb. For example, in Figure 4 (or the example E4), the word fix has higher frequency than the word read in our vocabulary, fix is then determined as the target verb.
- If multiple prepositions appear in a title, the first one appearing after the target verb (with one or more words in between) is determined as the target preposition. For instance, in the example E5 in Table 3, both words on and in are prepositions, but since the word on appears before in, on is then determined as the target preposition.
- If none of above applies for a vulnerability title, we conclude that it has no target word.

After recognizing the target verb and preposition for each vulnerability title, the keywords in between the two target words are extracted as the type for the vulnerability. However, as we list above, some cleaned titles may end up with only one target word or even no any target word. We handle those special titles as follows:

- If only the target verb exists, all words after the target verb will be regarded as the type keywords.
Table 5. The top 20 blockchain vulnerability types that affect at least ten vulnerabilities in our dataset.

| ID | Type                        | # Vulnerability Issues/PRs | Specific?* |
|----|-----------------------------|-----------------------------|------------|
|    |                             | All | Bitcoin | Ethereum | Monero | Stellar |          |
| T1 | Race Condition              | 77  | 14      | 48       | 10     | 5       | –         |
| T2 | Check/Validation            | 64  | 36      | 14       | 10     | 4       | –         |
| T3 | Resource Leak               | 47  | 24      | 12       | 9      | 2       | –         |
| T4 | Transaction Related         | 43  | 24      | 9        | 6      | 4       | –         |
| T5 | Deadlock                    | 36  | 16      | 13       | 6      | 6       | ✔         |
| T6 | Go Panic                    | 36  | 0       | 36       | 0      | 0       | –         |
| T7 | Block Related               | 34  | 9       | 21       | 4      | 0       | ✔         |
| T8 | Denial-of-Service           | 31  | 17      | 11       | 3      | 0       | –         |
| T9 | Peer/Node Related           | 28  | 12      | 11       | 3      | 2       | ✔         |
| T10| Sanity Check                | 28  | 11      | 3        | 13     | 1       | –         |
| T11| Overflow                    | 27  | 11      | 8        | 6      | 2       | –         |
| T12| Wallet Key/Password         | 25  | 12      | 6        | 7      | 0       | ✔         |
| T13| Uninitialized Read          | 19  | 14      | 0        | 5      | 0       | –         |
| T14| RPC Related                 | 16  | 9       | 5        | 2      | 0       | ✔         |
| T15| Out-of-Bound                | 14  | 9       | 4        | 1      | 0       | –         |
| T16| Off-by-One                  | 14  | 5       | 2        | 7      | 0       | –         |
| T17| Segfault                    | 13  | 13      | 0        | 0      | 0       | –         |
| T18| Memory Pool                 | 12  | 10      | 1        | 1      | 0       | –         |
| T19| Nil Pointer Deref           | 12  | 6       | 5        | 1      | 0       | –         |
| T20| Database Corruption         | 11  | 4       | 3        | 4      | 0       | –         |
| Sum|                             | 587 | 256     | 212      | 98     | 21      | –         |

* This column indicates whether vulnerabilities in one type are blockchain-specific, where ✔ means most vulnerabilities in this type are blockchain-specific and ◄ means some are blockchain-specific.

- If only the target preposition exists, all words before the target preposition will be treated as the type keywords.
- If no target word exists, the entire cleaned title becomes the type keywords.

**Clustering type keywords.** With the extracted type keywords, we aim to cluster them based on their semantic meaning rather than their appearance as a string of letters. Thus, after embedding all the keywords into the vector space using word2vec [44], we choose the Word Mover’s Distance (WMD) [34] as the similarity metric. Another reason for applying WMD is that it performs well on short sentences like our type keywords. Then, we calculate their pairwise similarity with WMD and generate a large similarity matrix.

The last step is to cluster the type keywords based on the similarity matrix. To reach an optimal clustering result, we tested four clustering algorithms: K-means [11], Gaussian Mixture [6], Agglomerative Clustering [5], and Affinity Propagation (AP) [22]. While the first three algorithms require a pre-defined number of clusters as the critical parameter, AP requires a damping factor. For the first three algorithms, we tried a wide range of cluster numbers from 25 to 225 with an interval of 2. For AP, we tried the damping factor from 0.5 to 1 with an interval of 0.01. We kept other parameters unchanged as default. After clustering with the given parameters, we computed the Silhouette Coefficient score [48] to determine the performance of the corresponding combination. As a result, Agglomerative clustering with 125 clusters was the best setting for our similarity matrix, which reached a coefficient score of 0.66.

5.2 Common Blockchain Vulnerability Types

According to Table 5, we obtain not only the traditional vulnerabilities, such as race condition and sanity check, but also blockchain-specific vulnerabilities. Indeed, among the top 20 vulnerability types, we find that seven of them are related to blockchains’ characteristics. In particular, the 130 (22.1%) vulnerabilities from five types (T4, T7, T9, and T12) are blockchain-specific, which are related to blockchains’ transaction, block, peer/node, and wallet key/password. Additionally, we have three more vulnerability types, T2, T14, and T20, that have some portions of their vulnerabilities related to blockchains’ features. The rest of 366 (62.4%) vulnerabilities are solely the traditional vulnerabilities, not specific to blockchains.
In the next three paragraphs, we explain three categories of these blockchain types: *specific*, *partially specific*, and *traditional*. For the patterns of blockchain-specific vulnerabilities, we will present them in §6.2.

**Blockchain-specific vulnerability types.** Since transactions, blocks, gas fees are the unique characteristics of blockchain systems, the type T4 and T7 record a large number of such new vulnerabilities. Examples are Bitcoin PR #8312 “Fix mempool DoS vulnerability from malleated transactions” and Ethereum PR #1354 “gpo non-existent block checks”. Moreover, as a peer-to-peer software by nature, blockchains could suffer from peer/node vulnerabilities. By inspecting 28 such vulnerabilities in the type T9, we find that they are mainly related to the unique P2P features in blockchains, such as header sync and block validation. Examples include Bitcoin PR #10345 “timeout for headers sync” and Ethereum issue #604 “SEC-41 Peer TD in NewBlockMsg not verified”. Lastly, blockchain systems often provide wallets to end users, which cause the new vulnerabilities related to wallet keys and passwords in the type T12. For example, Bitcoin PR #10308 describes the vulnerability patch of “[wallet] securely erase potentially sensitive keys/values”.

**Partially blockchain-specific vulnerability types.** We also observe three vulnerability types partially specific to blockchains, i.e., T2, T14, and T20. Specifically, 64 vulnerabilities in the type T2 performed various checks, e.g., error and length checks, and some of them checked blockchain-related properties. For example, Bitcoin issue #1167 “check for duplicate transactions earlier” for DoS prevention, and Ethereum PR #20546 “check propagated block malformation on reception”. In contrast, the type T14 and T20 fixed more traditional vulnerabilities related to RPC calls and database corruption (due to exceptional closing), with a few vulnerabilities directly related to blockchains. Examples of blockchain-related vulnerabilities are Ethereum PR #19401 “implement cli-configurable global gas cap for RPC calls” and Monero issue#706 “DB corruption” due to unfinished blockchain tasks.

**Traditional vulnerability types in blockchains.** Besides blockchain-specific vulnerabilities, Table 5 shows that 366 vulnerabilities are solely from the 13 traditional vulnerability types. The top types, such as race condition, deadlock, and denial-of-service, are more frequent probably because it is difficult for blockchain systems to avoid such vulnerabilities due to the sync among distributed nodes.

Besides the overall distribution of the top 20 vulnerability types in our dataset, Table 5 also lists the detailed distribution of these vulnerability types across different blockchain projects. We make the following several observations. First, Ethereum has more than half of the T1 (Race) vulnerabilities, much higher than the other three. After investigating all the race-related vulnerability issues/PRs, we identify that the Swarm [7] subsystem is the major cause. Specifically, Swarm is only available in Ethereum, and used for distributed storage and content distribution, such as node-to-node messaging, media streaming, decentralized database services for Dapps (decentralized applications). Second, we notice that T6 (Go Panic) only appears in Ethereum because only Ethereum is implemented in Go. Moreover, since Go is a memory-safe programming language, Ethereum has fewer memory-related (T13, T18) vulnerability issues/PRs than Bitcoin. Third, we find that Monero has the most number of T10 (Sanity Check) and T16 (Off-by-One) vulnerabilities, while Stellar has the least number of vulnerability types since it is relatively new.

**Key Takeaway in §5.2:** We successfully mapped 75.8% of the vulnerabilities into the clusters of different types and analyzed the top 20 types that affect at least ten vulnerabilities each. We identified four new vulnerability types that are directly related to blockchain transaction, block, peer/node, and wallet key/password. We also showed that traditional vulnerability types still hold 62%~78% of all the blockchain vulnerabilities. We further analyzed the vulnerability type differences across different blockchain projects.
6 CODE-LEVEL PATTERN ANALYSIS

At the third-level of our study, we perform the pattern analysis by analyzing vulnerability patch code. In particular, we focus on blockchain-specific vulnerability types (i.e., the seven types mentioned in §5.2) since the code patterns of traditional vulnerability types like race condition, deadlock, overflow, and uninitialized read are well-known (e.g., [15, 40, 54, 58]). In this section, we first propose our approach to summarizing patch code patterns in §6.1, and then present blockchain-specific code patterns in §6.2.

6.1 Generating and Clustering Code Change Signatures for Vulnerability Patterns

To obtain vulnerability code-level patterns, our objective is to put similar patch code changes into the same cluster so that analysts can summarize patterns from each cluster. To this end, we need an effective representation of code changes so that it keeps important semantic information yet ignores unimportant or noisy information. We call this representation the code change signature. Table 6 illustrates the evolution process from raw code hunks to their code fragments (i.e., contiguous lines of code) and the corresponding code change signatures using three examples. Taking the code in Table 6b and 6c as an example, both patches check whether the sender of a transaction is valid. However, if the variable name `senderAddr` is different, the similarity between their raw code fragment change (i.e., the syntactic changes indicated by F2 and F3) would be low. To capture the essential changes in patch code, we do not use the syntactic changes but their code change signatures like S2 and S3, the details of which will be illustrated during their generation.

Next, we introduce our approach to generating code change signatures and clustering them. Before these two major steps, we first clean up code hunks and turn them into fragments, and then align up the changed lines of code in each fragment.

Cleaning and splitting each code hunk into fragments. The raw code hunks we retrieved contain not only meaningful diff code but also test code, neighboring context (e.g., in-line and block comments, unchanged code lines, `#include` and `import` statements), and modification of none-code files (e.g., mark-down, JSON, and text files). Therefore, we first initiate a cleaning process [56] to keep only the actual diff code hunks and separate them into individual fragments by continuous `+' and `-` lines. Taking the code hunk in Table 6a as an example, it is separated into four code fragments, F1-1 (line 2-5), F1-2 (line 8-9), F1-3 (line 15-16), and F1-4 (line 19-21) after removing the neighboring context lines (i.e., line 1, 6-7, 10-14, 17-18, 22-23, and the comments in line 3 and 4).

Aligning up changed lines of code in each fragment. Before we generate each code fragment’s change signature based on deleted and added lines in it, we need to first pair up the changed lines of code since only some code fragments have one-to-one line change (i.e., at most one `-` line and one `+' line). For example, in Table 6, only the fragments F1-2 and F1-3 have one-to-one line change. For a multiple-line change in other fragments, we measure the edit distance similarity between each `-` line and all `+' lines and pair the one with the highest similarity. For instance, line 3 in Table 6c is paired with line 8 since it has the highest similarity with line 8 as compared with all the other lines. However, some lines could be simply deleted or added, causing their similarity with all other lines to be low. We handle this by not pairing the lines with the highest similarity of less than 0.5. As a result, line 3 in Table 6a will not be paired with line 5 due to the low similarity.

Generating the signatures of code changes. After determining the paired lines of code, we extract their syntactic changes [56] to generate the signatures with the following alterations:

- (Recognizing and marking the type of statements.) We first determine the control-flow statements by six reserved keywords, `if`, `for`, `while`, `return`, `throw`, and `defer`. If a control-flow statement is identified, we keep not only their type keyword but also their logical operators, such as `"\|\|"` in line 4 in Table 6b. If a statement does not contain any control-flow keyword,
we regard it as a function call if it includes a function or an assignment statement if it does not. For example, neither line 2 and line 4 in Table 6a have a control-flow keyword, but line 4 contains a function call cn_fast_hash(), so we regard line 4 as a function call; meanwhile, line 2 is regarded as an assignment statement.

• (Preserving the name only for a function call.) For function calls, we found that the function name itself is often enough to capture the statement nature despite parameter changes. Therefore, in code change signatures, we eliminate the function’s parameters and caller variables. For example, we eliminate the three parameters of cn_fast_hash() in Table 6a and keep its function name only. As a result, it is easy for the generated three signatures (S1-2/3/4) to be in the same cluster. The symbols for calling a function vary, including $(\text{fn}())$ and $(\text{fn} = \text{VAR})$. If a variable is an array, we further add one or more $\text{LEN}$ for strings; $\text{NIL}$ for nil, null, and none; $\text{BOL}$ for true and false; $\text{NUM}$ for numbers; $\text{TXT}$ for strings; $\text{LEN}/\text{SIZE}$ for size-related functions (e.g., $\text{len}()$, $\text{length}()$, $\text{size}()$, and $\text{sizeof}()$); and $\text{ERR}$ for error functions.

After all these alterations, we further insert a special change symbol “===” if the raw syntactic changes of a pair of ’-’ and ’+’ line have been abstracted into the signature, e.g., line 2 and 4 in

---

### Table 6. The evolution from raw code hunks to their code fragments and code change signatures.

**Example 1:** Monero commit 1d5e8f46.

```c
arc/crypto/tree-hash.c
@@ -82,23 +83,24 @@ void tree_hash(
 core/transaction_pool.go
@@ -116,7 +116,11 @@ func (pool *TxPool)
 chain/transaction_pool.go
@@ -65,5 +65,11 @@ func (pool *TxPool)
```
Table 7. 21 blockchain-specific patch code patterns obtained from the clustering result of 3,251 code fragments.

| Type          | Description                                                                 | Pattern (in the revised code change signature with some generalizations)                                                                                                                                                                                                 | Example* |
|---------------|------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------|
| Transaction   | Related                                                                      | P1 Check the transaction sender address                                                                                              | From(sender|address()|return|IsValid()|ERR() | E: #227 |
|               |                                                                              | P2 Check the size of transactions in a pool                                                                                           | getTotalSize()+size()|if | !=| MAX_STANDARD_TX_SIZE|return|BOL | B: #82079 |
|               |                                                                              | P3 Shuffle the transaction order, eitherwise, fingerprinting                                                                   | clear()|selected_coins()|shuffle()|push_back()       | B: #82079 |
|               |                                                                              | P4 Prevent the duplicated transaction                                                                                               | NUM(|REACHES()|insert()|return|ERR() | W: #11967 |
|               |                                                                              | P5 Prevent the malformed transaction                                                                                                 | if | !IsStandardTx()|return|BOL | B: #82112 |
|               |                                                                              | P6 Prevent the double-spent transaction (relay)                                                                                       | if | !IsValid()|return|BOL | B: #82164 |
| Block         | Related                                                                      | P7 Validate the new header not from an invalid block                                                                               | if | !IsValid()|return|BOL | S: #11531 |
|               |                                                                              | P8 Check the gas limit in a block header                                                                                              | GetGasLimit()|if | Eq()| NUM| return|ERR() | E: #8389 |
|               |                                                                              | P9 Check the block timestamp                                                                                                         | time()        | E: #9500 |
|               |                                                                              | P10 Validate some block fields (number and hash) not null                                                                           | GetBlocksByNumber()|()|Hash()|Eq()|NIL| break | E: #8354 |
|               |                                                                              | P11 Do not connect a corrupted block                                                                                                 | if | CorruptionPossible()|return|AbortNode()|ERR() | B: #12541 |
|               |                                                                              | P12 Prevent a malformed block to be propagated or forked                                                                            | !CalcHash()|GetHash()|Eq()|NI | break | E: #82064 |
|               |                                                                              | P13 Disconnect after the timeout of header synchronization                                                                           | if GetBlockTime()< GetAdjustedTime()-NUM|return|BOL | B: #10365 |
|               |                                                                              | P14 Disconnect invalid peers on the invalid chain                                                                               | if | GetHash()|Eq()|TolConnect|BOL | B: #11568 |
|               |                                                                              | P15 Strip the remote peer on an invalid or unverified TD                                                                             | if | GetBlock()|Eq()|# | B: #9074 |
| Wallet/Password| Related                                                                      | P16 Immediately wipe the memory for critical secret keys                                                                            | rclzklj()| MLSG_Gen()|memory_cleanse()|return | M: #2408 |
|               |                                                                              | P17 Try to keep the wallet address in testnet or memory                                                                               | generate()|if | !=| MAINNET|create_address_file()|ERR() | M: #35113 |
|               |                                                                              | P18 Try not stop asking for password when wallet-try                                                                                 | if | !=|ask_password()|B: #82175 |
|               |                                                                              | P19 Check the validity of Qosum set                                                                                                  | if | !IsQosumSetBane()|ERR()|if | throw|invalid_argument() | S: #2833 |
| Other Check   |                                                                              | P20 Enforce a gas cap of caller to protect against DoS                                                                                | if | RGasCap()|Eq()|NUM| return|ERR() | E: #19401 |
|               |                                                                              | P21 Avoid corruption due to unhandled blockchain tasks                                                                             | CRITICAL_REGION_LOCAL()|M: #7076 |

* This column lists one example issue/PR for each code pattern, where B, E, M, and S represent Bitcoin, Ethereum, Monero, and Stellar, respectively.

Table 6a. Otherwise, we do not insert the change symbol, as shown by line 8 and 9 in Table 6a and line 3 and 8 in Table 6c. After processing all the statements in a code fragment, we generate the fragment signature by concatenating each line-based signature.

**Clustering code change signatures.** As mentioned in the beginning of §6, we focus on clustering code change signatures from vulnerabilities of blockchain-specific types. Since the RPC-related and database corruption types (i.e., the type T14 and T20 in Table 5) have only one or two blockchain-specific vulnerabilities, there is no need to cluster their signatures. Eventually, our target is 3,251 code fragments from 194 vulnerabilities of the type T2, T4, T7, T9, and T12 (see §5.2). The clustering process is similar to that in §5.1. One major difference is that the WMD similarity is no longer applicable because code fragment signatures cannot be mapped to the token-based vector space. Therefore, we choose the Normalized Levenshtein distance [36] as the metric for calculating the similarity between code fragment signatures.

To find a suitable clustering algorithm here, we also tested the four algorithms in §5.1, i.e., K-means, Gaussian Mixture, Agglomerative Clustering, and Affinity Propagation (AP). For the first three algorithms that require a pre-estimation of the number of clusters, we compute the Silhouette Coefficient score in a wide range of cluster numbers, but the result is not satisfactory. Therefore, we choose AP as our code clustering algorithm since it does not require pre-setting the number of clusters and performs well with a gradual tuning of the damping factor to 0.78. Under this setting, we eventually obtain a total of 174 clusters for further pattern analysis.

### 6.2 Blockchain-specific Patch Code Patterns

After clustering code change signatures, we inspect all the clusters and generalize the code patterns from them. Table 7 lists the 21 evidently blockchain-specific vulnerability patterns. They are organized in seven categories by their types (see §5.2), and most check-related patterns have been categorized into the detailed types.

**Transaction-related patterns.** We have identified six patterns (P1–P6) related to blockchain transactions. They check the sender (P1), size (P2), and order (P3) of a transaction, and prevent duplicated (P4), malformed (P5), double-spent (P6) transactions. Specifically,

- **P1** checks a sender address function to guarantee non-null values with valid lengths. In Ethereum, the address function could be From() (#272) or Sender() (#195), as shown earlier in Table 6b and 6c. We also observed similar vulnerabilities in Bitcoin, but the function could be address() (#448) or addressFromUri() (#1002). Moreover, both cases use IsValid() to replace the raw null and length checks.
- P2 checks the maximum size of transactions allowed in a pool. In Bitcoin, the size of simultaneous transactions could be obtained from GetSerializeSize() (see the complete code of #2273 in Appendix A), while Ethereum directly retrieves the size via tx::Size(). This size is then checked with the maximum transaction size defined in blockchains; otherwise, DoS or CPU exhaustion attacks could occur.

- Besides the sender and length, the order of transactions could incur privacy risks like fingerprinting if not randomized. To address this, P3 is to clear the original order, shuffle it, and push_back the new order, as shown in Table 8a. Bitcoin #14897 does similarly by calling clear() and push_back() except that shuffle() is replaced by a random function.

- Both P4 and P5 check the blockchain structure to prevent duplicated or malformed transactions; otherwise, DoS could happen. For example, Bitcoin #1167 checks duplicated transaction IDs by BOOST EACH(), insert(), and size(), and returns DoS() if there are duplicates. The complete code is in Appendix A. Bitcoin #8312 similarly returns DoS() if non-standard transactions are identified via IsStandardTx().

- P6 prevents double-spent transaction relays via RelayableRespend(), which was checked in Bitcoin #4515 and #4450.

**Block-related patterns.** We identify another six patterns (P7–P12) related to blockchain blocks. As a basic blockchain unit, a block stores multiple transactions and will be appended to the chain according to the consensus protocols. However, vulnerabilities could happen if the header (P7), gas limit (P8), and timestamp (P9) of a new block is invalid, or if some block fields are not null (P10), or if a corrupted (P11) or malformed (P12) block is identified. Specifically:

- P7 validates that a newly appended block header is not from an invalid block. To do so, Bitcoin #11531 and #11487 invoke IsValid() , check block indexes in a while loop, mark invalid ones in an array via insert(), and then return DoS().

- P8 checks the gas limit in a block header, where gas is the fee for running smart contracts in Ethereum [14]. If a block exceeds the gas limit, it should not be added to the chain. For example, Ethereum #389 (see Appendix A) and #77 obtain the current gas limit via CalcGasLimit() and compare it with the limit in the block header.

- P9 checks whether the timestamp in a block is less than the current time(), such as Monero #5902 and Ethereum #1355. #389 in Table 11 in Appendix A also performs a similar check.
• **P10** validates the block fields like number and hash, and guarantees they are not null. For example, Ethereum #1354 and #19 check the block number via `GetBlockByNumber()` while Ethereum #19744 and #1939 check the hash via `Hash()`.

• Both **P11** and **P12** check the structure of a block to prevent a corrupted or malformed block being connected or forked. **P11** checks the corruption directly using a high-level API called `CorruptionPossible()` (see Table 12 in Appendix A), while **P12** performs the low-level checking via `CalcUncleHash()` and `DeriveSha()`, as shown in Table 8c.

**Peer/node-related patterns.** We also identify three patterns (P13–P15) related to peer/node synchronization and validation. Specifically, **P13** checks the time of block header synchronization, and if it is timed out, the node would disconnect. Table 8b shows such an example in Bitcoin #10345, where the timeout is checked via `GetBlockTime()`. A similar case is Bitcoin #5463 for the block download timeout. Additionally, **P14** and **P15** perform the validation of remote peers and drop them if they fail. For example, Bitcoin #11568 and #11446 in **P14** validate the hash of outbound peers via `GetHash()`. **P15**, on the other hand, checks the Ethereum-specific TD (Total Difficulty) field of a peer and guarantees the advertised TD actually deliverable, as in Ethereum #604 and #1451.

**Wallet-related patterns.** We further identify three patterns related to the blockchain wallet. First, since secret keys of a blockchain wallet are critical, **P16** immediately wipes the memory via `memwipe()` (Monero #4268) or `memory_cleanse()` (Bitcoin #10308) after generating some secrets, as shown in Table 8d. Second, the addresses in a wallet are also sensitive and should be kept in testnet or memory. For example, Monero #3315 in **P17** adds a `create_address_file` option for the address generating function `generate()` to create an address file only in the testnet environment. Similarly, Bitcoin #787 keeps the address table in memory and only writes to file when necessary. Third, a blockchain wallet requires users to always input passwords for critical operations. For example, Monero #4791 in **P18** performs such password checks via `ask_password()`.

**Other blockchain-specific patterns.** From P19 to P21, we summarize the last three kinds of blockchain-specific patterns. Specifically, **P19** checks the validity of a Stellar-specific concept called Quorum, which represents a set of nodes that are sufficient to reach an agreement in the Stellar network [39]. For example, Stellar #2233 (see Table 13 in Appendix A) and #2209 check the sanity of Quorum via `isQuorumSetSane()`. **P20** is a RPC-related pattern, which restricts the gas cap of RPC calls. If the requested gas exceeds the cap limit via `RPCGasCap()`, the caller should be warned (see Ethereum #19401 in Appendix A). The last pattern, **P21**, asks a blockchain client to gracefully shutdown itself when there are unfinished block synchronization and processing. This can be done by setting a global blockchain lock via `CRITICAL_REGION_LOCAL()`, as shown in Monero #706.

**Demonstrating the usage of these vulnerability patterns.** The major usage is to facilitate the detection of similar vulnerabilities in other blockchain projects. To demonstrate that, we have leveraged the aforementioned patterns to identify 23 previously unknown vulnerabilities in six popular blockchains, including the rank #10 Dogecoin, #50 Bitcoin SV, #63 Dash, #65 Zcash, #95 Ravencoin, and #102 Horizen. We also reported all these vulnerabilities to their corresponding vendors and offered them fix suggestions. Most of our vulnerability reports have been confirmed. However, since these vulnerabilities are still under patching, we cannot disclose the details here, and will make them available at an appropriate timing for research ethics.

---

**Key Takeaway in §6.2:** We identified 21 blockchain-specific vulnerability patterns from 174 clusters of code change signatures generated from 3,251 code fragments. These patterns check unique blockchain attributes, e.g., the sender address, transaction order, block header, and gas limit, and validate various blockchain statuses during node synchronization, peer validation, wallet and database operations. We further demonstrated that they can be used to detect similar vulnerabilities in other top blockchains.
7 RELATED WORK

In this section, we go through some related works on blockchain vulnerabilities, empirical vulnerability studies, and mining-based vulnerability detection.

Blockchain vulnerability research. Existing blockchain security studies mainly focus on smart contract vulnerability detection and transaction- or network-level analysis of the blockchains. For **smart contract vulnerability detection**, both static and dynamic program analysis tools have been proposed. For instance, Oyente [42], Zeus [29], Securify [52], Gigahorse [25], and ETHBMC [21] detected vulnerable smart contracts via symbolic execution, while ContractFuzzer [27] and ConFuzzius [51] used fuzzing inputs to detect smart contract vulnerabilities, and Sereum [47] and SODA [19] monitored run-time contract execution to detect on-chain attacks in modified EVMs. Moreover, learning-based tools, such as SmartEmbed [24], ESCORT [41], and AMEVulDetector [38] were also recently invented. For **transaction-level analysis**, Karame et al. [31] analyzed the double-spending resilience of Bitcoin fast payments. Chen et al. [20] performed a systematic study of Ethereum transactions via graph analysis. TxSpector [63] studied Ethereum transactions by replaying historical transactions and recording EVM bytecode-level traces. DeFiPoser [66] proposed two automatic methods for discovering profit-generating transactions in DeFi protocols just in time. For **network-level analysis**, Apostolaki et al. [10] analyzed routing attacks by hijacking BGP prefixes and demonstrated that such attacks could delay the propagation of blocks without being detected. Gao et al. [23] showed that by power adjusting and bribery racing, attackers could increase their mining rewards.

Empirical vulnerability studies. To the best of our knowledge, in the specific topic of the empirical study on blockchain systems, there is only one such work [53], and they focused on general kinds of bugs (e.g., semantic and GUI bugs), with only 18 security bugs covered and four analyzed. If we loosen the vulnerability research domain to other systems, much more empirical studies have been conducted. Back in 2003, Chen et al. [18] performed an exploration of memory-corruption vulnerabilities and modeled them by a finite-state machine. On Linux vulnerabilities, Chen et al. [16] first conducted a large-scale analysis on 100+ Linux kernel vulnerabilities in 2011, and You et al. [60] proposed a semantic-based approach on analyzing Linux CVE reports and git logs to generate proof-of-concept exploits. Wu et al. [56] and Zhang et al. [64] both targeted Android system vulnerabilities with different perspectives. While Wu et al. [56] conducted a source code-level analysis on Android security patches, Zhang et al. [64] focused on the patching propagation behaviors in the entire Android kernel ecosystem. Web security is also a hot research area. Zhao et al. [65] studied web vulnerability discovery ecosystems based on two popular bounty programs.

Mining-based vulnerability detection. Code clone detection is a long-standing research topic in the software engineering area, and it has been used by the security community for mining vulnerable code fragments (e.g., [26, 33, 57, 59]). Existing approaches are mainly based on detecting duplicated token subsequences or identifying exact or similar subtrees in abstract syntax tree (AST) representations. For **token-based** approaches, CCFinder [30], CP-Miner [37], and ReDeBug [26] are the representative work, all of which first split code into token sequences and then calculate the similarity of the code tokens for multilingual clone detection. More recently, a token-based approach, VUDDY [33], generated code fingerprints via abstraction and normalization to speed up code clone detection. For **tree-based** approaches, e.g., DECKARD [28] and CloneDR [12], they considered code’s structural information by generating ASTs and embedding them into a vector space for similarity comparison. In addition, there are also approaches based on graphs and program slicing, such as Code Property Graph [59] and MVP [57] that extract graph and slice features for vulnerability matching.
8 CONCLUSION

In this paper, we conducted the first empirical study of blockchain system vulnerabilities and their security patches using four representative blockchains, Bitcoin, Ethereum, Monero, and Stellar. To enable this study, we proposed a vulnerability filtering framework to effectively identify 1,037 vulnerabilities and their 2,317 patches from 34,245 issues/PRs and 85,164 commits on GitHub. Based on this unique dataset, we performed three levels of analyses, namely file-level vulnerable module categorization, text-level vulnerability type clustering, and code-level vulnerability pattern analysis. Our analysis revealed three key findings of blockchain system vulnerabilities, including (i) the modules related to consensus, wallet, and networking are highly susceptible, each with over 200 issues; (ii) around 70% of blockchain vulnerabilities are in traditional types, but we also identify four new types specific to blockchains; and (iii) we are able to obtain 21 blockchain-specific vulnerability patterns that check unique blockchain attributes and validate various blockchain statuses, and show that they can be applied to discover 23 similar vulnerabilities in other top blockchains, such as Dogecoin and Bitcoin SV. In the future, we will continue to detect blockchain system vulnerabilities based on the knowledge base established in this paper.

APPENDIX

A SUPPLEMENTARY PATCH EXAMPLES

In this section, we provide eight supplementary patch code examples for the contents in §6.2. Specifically, Table 9, 10, 11, 12, 13, and 14 describe the patch code examples for the pattern P2, P4, P8, P11, P19, and P20, respectively.

Table 9. A patch code example of P2, Bitcoin #2273

```c
bool CTransaction::IsStandard() const {
    // Extremely large transactions with lots of inputs can cost the network
    // almost as much to process as they cost the sender in fees, because
    // computing signature hashes is O(inputs*txsize). Limiting txns
    // to MAX_STANDARD_TX_SIZE mitigates CPU exhaustion attacks.

    unsigned int sz = this->GetSerializeSize(...);
    if (sz >= MAX_STANDARD_TX_SIZE)
        return false;
    ...}
```

Table 10. A patch code example of P4, Bitcoin #1167

```c
bool CBlock::CheckBlock() const {
    // Check for duplicate txids. This is caught by ConnectInputs(),
    // but catching it earlier avoids a potential DoS attack:
    set<uint256> uniqueTx;
    BOOST_FOREACH(const CTransaction& tx, vtx)
        uniqueTx.insert(tx.GetHash());
    if (uniqueTx.size() != vtx.size())
        return DoS(100, error("CheckBlock() : duplicate transaction"));
    ...}
```

Table 11. A patch code example of P8, Ethereum #389

```c
func (sm *BlockProcessor) ValidateBlock(block, parent *types.Block) error {
    expl := CalcGasLimit(parent, block)
    if expl.Cmp(block.Header().GasLimit) != 0 {
        return fmt.Errorf("GasLimit check failed for block", ...)
    }
    if block.Time() < parent.Time() {
        return ValidationError("Block timestamp not after prev block", ...)
    }
    ...}
```

Table 12. A patch code example of P11, Bitcoin #12561

```c
bool CChainState::ConnectBlock(const CBlock& block, CValidationState& ...) {
    ...
    if (!CheckBlock(block, state, chainparams.GetConsensus(), ...)) {
        if (state.CorruptionPossible()) {
            // We don't write down blocks to disk if they may have been
            // corrupted...
            return AbortNode(state, ...);
        }
    }
    ...}
```

Table 13. A patch code example of P19, Stellar #2233

```c
Config::validateConfig(bool mixed) {
    ...
    if (!isQuorumSetSane(QUORUM_SET, !UNSAFE_QUORUM))
        LOG(FATAL) << fmt::format("Invalid QUORUM_SET: check nesting, "
                                 "duplicate entries and thresholds (must be "
                                 "between and 100)".
                                 "UNSAFE_QUORUM ? 1 : 51);"
        throw std::invalid_argument("Invalid QUORUM_SET");
    ...
}
```

Table 14. A patch code example of P20, Ethereum #19401

```c
func (s *PublicBlockChainAPI) EstimateGas(ctx context.Context, ...) ... {
    if gasCap := s.b.RPCGasCap(); gasCap != nil {
        if *args.Gas.Cmp(gasCap) > 0 {
            log.Warn("Applying cap on gas, caller requested amount above ...")
            newGas := hexutil.Uint64(gasCap.Uint64())
            *args.Gas = newGas
            ...
        }
    return DoEstimateGas(ctx, s.b, args, rpc.PendingBlockNumber)
}
```
REFERENCES

[1] 2019. Bitcoin Core 0.11 (ch 1): Overview. https://en.bitcoin.it/wiki/Bitcoin_Core_0.11_(ch_1):_Overview.
[2] 2020. Bitcoin Core: The Reference Implementation. https://cypherpunks-core.github.io/bitcoinbook/ch03.html.
[3] 2020. NLTK: Natural Language Toolkit. https://www.nltk.org/.
[4] 2020. spaCy: Industrial-Strength Natural Language Processing. https://spacy.io/.
[5] 2021. Agglomerative Clustering. https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html.
[6] 2021. Gaussian Mixture Models. https://scikit-learn.org/stable/modules/generated/sklearn.mixture.GaussianMixture.html.
[7] 2021. Swarm. https://github.com/ethersphere/swarm.
[8] Muneeb Ali, Jude Nelson, Ryan Shea, and Michael J Freedman. 2016. Blockstack: A global naming and storage system secured by blockchains. In USENIX ATC.
[9] Elli Androulaki, Artem Barger, Vita Bortnikov, Konstantinos Christidis, Angelo De Caro, David Eneyart, Christopher Ferris, Gennady Laventman, and Yacov Manevich. 2018. Hyperledger Fabric: A distributed operating system for permissioned blockchains. In Proc. ACM EuroSys.
[10] Maria Apostolaki, Aviv Zohar, and Laurent Vanbever. 2017. Hijacking Bitcoin: Routing Attacks on Cryptocurrencies. In Proc. IEEE Symposium on Security and Privacy.
[11] David Arthur and Sergei Vassilvitskii. 2007. K-Means++: The Advantages of Careful Seeding. In Proc. ACM SODA.
[12] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant’Anna, and Lorraine Bier. 1998. Clone Detection Using Abstract Syntax Trees. In Proc. ACM ICSE.
[13] Juan Benet. 2014. IPFS-content addressed, versioned, p2p file system. CoRR arXiv abs/1407.3561 (2014).
[14] Vitalik Buterin. 2014. A next-generation smart contract and decentralized application platform. white paper (2014).
[15] Yan Cai and Wing-Kwong Chan. 2014. Magiclock: Scalable detection of potential deadlocks in large-scale multithreaded programs. IEEE Transactions on Software Engineering (2014).
[16] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. 2011. Linux Kernel Vulnerabilities: State-of-the-Art Defenses and Open Problems. In Proc. ACM APSys.
[17] Mengjie Chen, Daoyuan Wu, Xiao Yi, and Jianliang Xu. 2021. AGChain: A Blockchain-based Gateway for Permanent, Distributed, and Secure App Delegation from Existing Mobile App Markets. CoRR arXiv abs/2101.06454 (2021).
[18] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, Yuxing Tang, Xiaodong Lin, and Xiaosong Zhang. 2017. SODA: A Generic Online Detection Framework for Smart Contracts. In Proc. ISOC NDSS.
[19] Ting Chen, Yuxiao Zhu, Zhihao Li, Jiachi Chen, Xiaoqi Li, Xiapu Luo, Xiaodong Lin, and Xiaosong Zhang. 2018. Understanding Ethereum via Graph Analysis. In Proc. IEEE INFOCOM.
[20] Joel Frank, Cornelius Aschermann, and Thorsten Holz. 2020. ETHBMC: A Bounded Model Checker for Smart Contracts. In Proc. USENIX Security.
[21] Brendan J. Frey and Delbert Dueck. 2007. Clustering by Passing Messages Between Data Points. Science (2007).
[22] Shang Gao, Zecheng Li, Zhe Peng, and Bin Xiao. 2019. Power Adjusting and Bribery Racing: Novel Mining Attacks in the Bitcoin System. In Proc. ACM CCS.
[23] Zhipeng Gao, Lingxiao Jiang, Xin Xia, David Lo, and John Grundy. 2020. Checking Smart Contracts with Structural Code Embedding. IEEE Transactions on Software Engineering (2020).
[24] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: Thorough, Declarative Decomplation of Smart Contracts. In Proc. ACM ICSE.
[25] Jiyoung Kang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In Proc. IEEE Symposium on Security and Privacy.
[26] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In Proc. ACM ASE.
[27] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In Proc. ACM ICSE.
[28] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In Proc. ISOC NDSS.
[29] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A Multilingualistic Token-based Code Clone Detection System for Large Scale Source Code. IEEE Transactions on Software Engineering (2002).
[30] Ghassan O. Karame, Elli Androulaki, and Srdjan Capkun. 2012. Double-Spending Fast Payments in Bitcoin. In Proc. ACM CCS.
[32] Olga Kharif. 2021. Crypto Market Cap Surpasses $2 Trillion After Doubling This Year. https://www.bloomberg.com/news/articles/2021-04-05/crypto-market-cap-doubles-past-2-trillion-after-two-month-surge.

[33] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In Proc. IEEE Symposium on Security and Privacy.

[34] Matt J. Kusner, Yu Sun, Nicholas I. Kolkin, and Kilian Q. Weinberger. 2015. From Word Embeddings to Document Distances. In Proc. IMLS ICML.

[35] Frank Li and Vern Paxson. 2017. A Large-Scale Empirical Study of Security Patches. In Proc. ACM CCS.

[36] Yujian Li and Bi Liu. 2007. A normalized Levenshtein Distance metric. IEEE Transactions On Pattern Analysis and Machine Intelligence (2007).

[37] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In Proc. USENIX OSDI.

[38] Zhenguang Liu, Peng Qian, Xiang Wang, Lei Zhu, Qinming He, and Shouling Ji. 2021. Smart Contract Vulnerability Detection: From Pure Neural Network to Interpretable Graph Feature and Expert Pattern Fusion. In Proc. IJCAI.

[39] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafal Malinowsky, and Jed McCaleb. 2019. Fast and Secure Global Payments with Stellar. In Proc. ACM SOSP.

[40] Mangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nünberger, Wenke Lee, and Michael Backes. 2017. Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying. In Proc. ISOC NDSS.

[41] Oliver Lutz, Huili Chen, Mohammad Fereidooni, Christoph Sendner, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. 2021. ESCORT: Ethereum Smart Contract Vulnerability Detection using Deep Neural Network and Transfer Learning. CoRR abs/2103.12607 (2021).

[42] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In Proc. ACM CCS.

[43] Deepak Maram, Harjasleen Malvai, Fan Zhang, Nerla Jean-Louis, Alexander Frolov, Tyler Kell, Troye Lobban, Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In Proc. ACM CCS.

[44] Matt J. Kusner, Yu Sun, Nicholas I. Kolkin, and Kilian Q. Weinberger. 2015. From Word Embeddings to Document Distances. In Proc. NIPS.

[45] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. white paper (2008).

[46] Shen Noether. 2015. Ring Signature Confidential Transactions for Monero. IACR Cryptol. ePrint Arch. (2015).

[47] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. 2019. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. In Proc. ISOC NDSS.

[48] Peter J. Rousseuw. 1987. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. J. Comput. Appl. Math. (1987).

[49] Bo Tang, Hongjuan Kang, Jingwen Fan, Qi Li, and Ravi S. Sandhu. 2019. IoT Passport: A Blockchain-Based Trust Framework for Collaborative Internet-of-Things. In Proc. ACM SACMAT.

[50] Yuan Tian, Julia L. Lawall, and David Lo. 2012. Identifying Linux Bug Fixing Patches. In Proc. ACM ICSE.

[51] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. 2021. ConFuzzius: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts. In Proc. IEEE European Symposium on Security and Privacy.

[52] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In Proc. ACM CCS.

[53] Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. 2017. Bug Characteristics in Blockchain Systems: A Large-Scale Empirical Study. In Proc. ACM MSR.

[54] Zhilong Wang, Xuhua Ding, Chengbin Pang, Jian Guo, Jun Zhu, and Bing Mao. 2018. To detect stack buffer overflow with polymorphic canaries. In Proc. DSN.

[55] Sam M Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William J Knottenbelt. 2021. SoK: Decentralized Finance (DeFi). CoRR arXiv abs/2101.08778 (2021).

[56] Daoyuan Wu, Debin Gao, Eric K. T. Cheng, Yichen Cao, Jintao Jiang, and Robert H. Deng. 2019. Towards Understanding Android System Vulnerabilities: Techniques and Insights. In Proc. ACM AsiaCCS.

[57] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binhong Liu, Yang Liu, Wei Huo, Wei Zou, and Wenchang Shi. 2020. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. In Proc. USENIX Security.

[58] Meng Xu. 2020. Finding Race Conditions in Kernels: The Symbolic Way and the Fuzzy Way. Ph.D. Dissertation. Georgia Institute of Technology.

[59] Fabian Yamauchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In Proc. IEEE Symposium on Security and Privacy.

[60] Wei You, Peiyuan Zong, Kai Chen, Xiaofeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. SemFuzz: Semantics-Based Automatic Generation of Proof-of-Concept Exploits. In Proc. ACM CCS.
[61] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. 2016. Town Crier: An authenticated data feed for smart contracts. In ACM CCS.

[62] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. 2020. DECO: Liberating web data using decentralized oracles for TLS. In Proc. ACM CCS.

[63] Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. 2020. TXSPECTOR: Uncovering Attacks in Ethereum from Transactions. In USENIX Security.

[64] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. 2021. An Investigation of the Android Kernel Patch Ecosystem. In Proc. USENIX Security.

[65] Mingyi Zhao, Jens Grossklags, and Peng Liu. 2015. An Empirical Study of Web Vulnerability Discovery Ecosystems. In Proc. ACM CCS.

[66] Liyi Zhou, Kaihua Qin, Antoine Cully, Benjamin Livshits, and Arthur Gervais. 2021. On the Just-In-Time Discovery of Profit-Generating Transactions in DeFi Protocols. In Proc. IEEE Symposium on Security and Privacy.

[67] Yaqin Zhou and Asankhaya Sharma. 2017. Automated Identification of Security Issues from Commit Messages and Bug Reports. In Proc. ACM FSE.