Bug Characteristics in Blockchain Systems: A Large-Scale Empirical Study

Zhiyuan Wan[1], David Lo[2], Xin Xia[3‡], and Liang Cai[1]
[1]College of Computer Science and Technology, Zhejiang University, Hangzhou, China
[2]School of Information Systems, Singapore Management University, Singapore
[3]Department of Computer Science, University of British Columbia, Canada
{wanzhiyuan, leoncai}@zju.edu.cn, davidlo@smu.edu.sg, xxia02@cs.ubc.ca

***Abstract*—Bugs severely hurt blockchain system dependability. A thorough understanding of blockchain bug characteristics is required to design effective tools for preventing, detecting and mitigating bugs. We perform an empirical study on bug characteristics in eight representative open source blockchain systems. First, we manually examine 1,108 bug reports to understand the nature of the reported bugs. Second, we leverage card sorting to label the bug reports, and obtain ten bug categories in blockchain systems. We further investigate the frequency distribution of bug categories across projects and programming languages. Finally, we study the relationship between bug categories and bug fixing time. The findings include: (1) *semantic* bugs are the dominant runtime bug category; (2) frequency distributions of bug types show similar trends across different projects and programming languages; (3) *security* bugs take the longest median time to be fixed; (4) 35.71% *performance* bugs are fixed in more than one year; *performance* bugs take the longest average time to be fixed.**

I. Introduction

Bitcoin has emerged as the first widely-deployed, decentralized cryptocurrency, which sparks hundreds of cryptocurrencies. They have attracted a lot of attention from the financial and public regulatory sectors. The overall capitalization of cryptocurrencies reaches 12 billion USD as of October 2016 [1]; the venture capital investment reaches 1 billion USD as of October 2016 [2]. The core technological innovation powering cryptocurrencies is a distributed ledger known as blockchain. Blockchain technology provides an open, decentralized and fault-tolerant transaction mechanism. It promises to become the infrastructure for a new generation of Internet interaction, including anonymous online payment [3], remittance, and transaction of digital assets [4]. Ongoing work explores smart digital contracts, enabling anonymous parties to programmatically enforce complex agreements [5, 6].

Bugs severely hurt the dependability of blockchain systems. For instance, an attacker exploited a bug in the DAO project - a project that is launched on the Ethereum blockchain[1]. Consequently, the attacker succeeded to put approximately 60 million dollars under her control by 18th June 2016, until the hard-fork of the blockchain discarded the transaction involved in the attack.

Understanding bug characteristics could help to design effective tools for preventing, detecting and mitigating bugs. Previous work performs empirical studies on software bugs to understand bug characteristics [7–14]. These studies provide knowledge for guiding design of bug detection tools, triaging bug reports, locating bug locations, suggesting possible bug fixes, gauging testing and debugging costs, measuring software quality, and helping to monitor and manage development processes.

To the best of our knowledge, bug characteristics of blockchain systems have not been studied. Thus we would like to perform an empirical study on bug characteristics of blockchain systems, and answer new research questions (RQ) about blockchain systems. We analyze the bug reports of eight representative open source blockchain systems with large market capitalization[2], whose public repositories are on GitHub: (1) bitcoin/bitcoin, a cryptocurrency and a payment system. (2) ethereum/go-ethereum, an official Go implementation of the Ethereum protocol[3]. (3) ethereum/mist, an Ethereum browser, which offers an overall view of the Ethereum blockchain, and tools to interact with the blockchain components. (4) dogecoin/dogecoin, a cryptocurrency. (5) ethereum/cpp-ethereum, an Ethereum C++ client. (6) ripple/ripple-lib, a JavaScript API for interacting with Ripple in Node.js. Ripple is a real-time gross settlement system, currency exchange and remittance network; it is built upon blockchain technology as consensus ledge. (7) steemit/steem, an experimental proof of work blockchain with an unproven consensus algorithm; steem enables the social media platform steemit to pay the contributors for their posts. (8) AugurProject/augur, a decentralized prediction market platform built on Ethereum.

To ensure correct results, we only study *closed* bug reports. This is because root causes of *open* bug reports are often unknown. We first manually examine 1,108 closed *bug reports* and identify 946 unique *bugs*. The number of bugs is smaller than the number of bug reports, because some bug reports are invalid and duplicate. We then apply card sorting to classify bugs and investigate the nature of these bugs in blockchain systems.

Our study aims to answer a number of research questions: What categories of bugs appear in blockchain systems? Do the frequency distributions of bug types show similar trends across

‡Corresponding author.

[1]http://www.coindesk.com/understanding-dao-hack-journalists/

[2]https://coinmarketcap.com/. We refer to the market capitalization of October 10, 2016

[3]https://github.com/ethereum/wiki/wiki/White-Paper

different blockchain projects? Do the frequency distributions of bug types vary substantially across different programming languages? How long does it take to fix various categories of bugs? We answer the above questions by performing manual analysis on bug reports and their corresponding pull request and commits for bug fixing.

Our contributions are as follows:

1) We are the first to perform a large-scale empirical study of bugs in blockchain systems.
2) We manually categorize 946 bugs in the eight studied blockchain systems into various types by using card sorting.
3) We analyze the frequency distributions of bug types across different projects and programming languages.
4) We investigate the relationship between bug categories and bug fixing time.

The rest of the paper is structured as follows. In Section II we describe our research questions, data collection, dataset and methodology. Section III, IV, and V present our empirical study results. Section VI discusses the implications of our study and the threats to validity. Section VII briefly reviews the related work. Section VIII draws conclusions and presents future work.

II. Empirical Study Setup

A. Research Questions

RQ1. What categories of bugs appear in blockchain systems?

Bug categorization can help to understand the weakness of blockchain systems. More effort could be put into addressing the dominant bug category.

RQ2. Are the frequency distributions of bug types similar across different blockchain projects?

Different blockchain projects are developed under different requirements and intend to accomplish different tasks. Do they show similar frequency distribution of bug types? If the frequency distributions of bug types show similar trends across projects, we can rank the bug types from most to least common, and then summarize characteristics of each bug type.

RQ3. Do the frequency distributions of bug types vary substantially across different programming languages?

Programming languages have specific features classified by programming paradigm (procedural, scripting or functional), compilation class (static or dynamic), type class (strong or weak), and memory class (managed or unmanaged) [15]. For example, C++ is a *procedural*, *static* and *weakly* typed language with unmanaged memory types. We wonder if different programming languages would affect the frequency distribution of bug types in blockchain systems.

RQ4. How long does it take to fix various types of bugs?

We intend to investigate the relationship between bug type and bug fixing time. The result may indicate that more attention should be paid to a specific bug type.

B. Data Collection

Retrieving representative blockchain projects. We initially investigate top 20 cryptocurrencies by market capitalizations [1] and top 10 venture capital blockchain companies [2]. For the cryptocurrencies, we follow the official links provided by the `CoinMarketCap` website; for the venture capital companies, we search for the companies, which often publish their software on code-hosting platforms (e.g., GitHub, Bitbucket, and Google Code).

We initially find 19 public accessible, open source blockchain systems whose source code is hosted on GitHub. Then we exclude those systems: a) that fork from bitcoin project, b) whose issue repositories have no bug tag, c) that have no closed issues, and d) whose durations between the first and last bug reports are less than 2 weeks. Finally, we get eight blockchain systems as shown in Table I.

Retrieving bug reports and related information. We obtain bug repositories including the issue title, body and comments of the eight blockchain systems through GitHub API. A single bug can be fixed by one pull request or one/multiple commits. GitHub repositories store the linkages between reported bugs and pull requests or code commits that fix the bugs. To facilitate further analysis, we also obtain pull requests and commits.

Identifying primary languages. Multiple languages are often used to develop a project. Assigning a single language to a project is difficult. GitHub uses `Linguist` [16] library to measure the language distribution of each GitHub project repository. Linguist first identifies the languages by the extension of source files, and then counts the number of source files with different extensions. The language with the maximum number of source files is assigned as *primary language* of the project. GitHub Archive stores the *primary language* information.

Calculating bug fixing duration. By using GitHub API, we obtain the creation date (`created_at`) and closing date (`closed_at`) of each closed bug report from the issue repositories. The fixing duration of a bug starts at its creation date and ends at its closing date.

C. Dataset

bitcoin/bitcoin is an open source software which enables the use of the cryptocurrency bitcoin. Bitcoin is the most popular decentralized cryptocurrency. The system under Bitcoin is peer-to-peer – users can transact without an intermediary. Blockchain records those transactions as a public distributed ledger. The repository has been created on GitHub since Dec 2010. Its latest release on 14 Oct 2016 is version 0.13.1.

ethereum/go-ethereum is an official golang implementation of the Ethereum protocol. Ethereum is a public blockchain-based distributed computing platform, providing a decentralized virtual machine to execute peer-to-peer smart contracts. Ethereum uses a cryptocurrency called *ether*, which is the second largest cryptocurrency. The repository has been created on GitHub since Dec 2013. Its latest release on 15 Oct 2016 is version 1.4.18.

ethereum/mist is an Ethereum browser. It offers an overall view of the Ethereum blockchain and tools to interact with

TABLE I: Blockchain systems studied. Open https://github.com/<GitHub Repository> for details.

GitHub Repository	Stargazers	SLOC	# Closed BR	Duration	Contributors	Releases	Language
bitcoin/bitcoin	10356	80,406	401	5.77 years	402	165	C++
ethereum/go-ethereum	2393	455,464	89	2.36 years	77	100	Go
ethereum/mist	1270	21,640	84	1.14 years	37	32	JavaScript
dogecoin/dogecoin	1155	66,270	27	2.16 years	298	32	C++
ethereum/cpp-ethereum	758	105,624	272	2.68 years	90	227	C++
ripple/ripple-lib	429	10,217	19	1.77 years	44	117	JavaScript
steemit/steem	207	80,927	19	0.32 years	15	37	C++
AugurProject/augur	157	266,589	197	1.27 years	15	5	JavaScript

the blockchain components. The repository has been created on GitHub since Jun 2015. Its latest release on 26 Oct 2016 is version 0.8.7.

dogecoin/dogecoin is an open source peer-to-peer crytocurrency. The Dogecoin blockchain is a public decentralized ledger of all Dogecoin transactions. The repository has been created on GitHub since Dec 2013. Its latest release on 22 Mar 2016 is version 1.10.1-alpha-1.

ethereum/cpp-ethereum is the Ethereum C++ client. The repository has been created on GitHub since Dec 2013. Its latest release on 7 Aug 2016 is version `untagged-1d50efdb2f43825a1810`.

ripple/ripple-lib is a JavaScript API for interacting with Ripple in Node.js. Ripple is a real-time gross settlement system, currency exchange and remittance network. It is built upon blockchain technology as consensus ledge. Its own cryptocurrency *XRP* is the third-largest cryptocurrency by market capitalization.[4] The repository has been created on GitHub since Apr 2013. Its latest release on 30 Sep 2016 is version 0.17.3.

steemit/steem is an experimental proof of work blockchain with an unproven consensus algorithm. Steem enables the social media platform steemit to pay the contributors for their posts. The repository has been created on GitHub since Mar 2016. Its latest release on 2 Nov 2016 is version 0.15.0.

AugurProject/augur is an open source and decentralized prediction market platform built on Ethereum. It leverages the blockchain's functionality as well as game theory and financial incentives to make more accurate predictions about future events.[5] The repository has been created on GitHub since Feb 2015. Its latest release on 30 Apr 2016 is version 1.0.0.

D. Methodology

We analyze snapshots of GitHub issue repositories of the eight blockchain systems dated up to 7 Nov 2016. We focus on *closed* bug reports from the issue repositories, i.e., *closed* issues with tag *bug*. We exclude *open* bug reports because they are not fixed and may not have enough information for our analysis, or they may not even be bugs. TABLE I shows the number of closed bug reports for the eight blockchain projects. The durations between the first and last bugs are shown in Column "Duration".

We manually look into the title, body, comments of each bug report. Such information is often short and ambiguous.

[4]http://coinmarketcap.com

[5]http://augur.strikingly.com/blog/the-pressing-need-for-augur

TABLE II: Classification scheme.

Category	Description	Abbreviation
Memory	Bugs caused by improper handling of memory objects.	Mem
Concurrency	Synchronization problems among the concurrent tasks in concurrent programs?[19], including data races and deadlocks.	Con
Performance	Bugs that make a system perform abnormally in terms of responsiveness and stability under a normal workload.	Perf
Security	Vulnerabilities that cause damage to the software or the inforamtion on them, as well as the services they provide.	Sec
Environment and Configuration	Errors in dependent libraries, underlying operating systems, or non-code that affects functionality.	EnvConf
GUI	Graphical user interface errors, including incorrect font, alignment, and button size.	GUI
Build	Link and compilation errors.	Build
Compatibility	A software cannot normally run on a particular CPU architecture, operating system, or Web browser, etc.	Comp
Hard Fork	Errors due to radical changes to the protocol that makes previous valid blocks/transactions invalid (or vice-versa).	HardFork
Semantic	Inconsistencies with the requirements or the programmers' intention that do not belong to the categories above.	Sem

So we further exploit related pull request and commit logs, similar to previous work [14, 17]. We follow the card sorting approach to label the bug reports [18].

Step 1: Card Sorting. We create one card for each of the bug reports. The card contains bug information from several data sources, e.g., bug report title, body and comments, pull requests, and commits. The first author and one graduate student from Zhejiang University jointly figure out the labels for the bug reports. The detailed steps are as below:

Iteration 1. We first randomly pick the ethereum/go-ethereum project and manually inspect the bug reports. Then we sort the bug reports into distinct bug sets according to their root causes. The root causes of some bug reports are unclear and we omit them from our card sort. Finally, we discuss each bug set and name it by referring to the categories that were defined in Tan et al.'s study [14]. The resulting classification scheme contains 6 categories as shown in TABLE II (except *GUI*, *build*, *compatibility* and *hard fork* categories).

Iteration 2. We find that *semantic* bugs account for a large proportion in project ethereum/go-ethereum. So we further classify *semantic* bugs into 9 subcategories shown in Table III. The subcategories are based on the *semantic* bug subcategories in [14]. In addition, we create two new subcategories *input* and *output*.

Iteration 3. First, we manually inspect bug reports in oth-

TABLE III: Subcategories of *semantic* bugs.

Category	Subcategory	Description
***Semantic* Bug**	Missing Features	A feature is supposed to be but is not implemented.
	Missing Cases	A case in a functionality is not implemented.
	Corner Cases	Some boundary cases are considered incorrectly or ignored.
	Wrong Control Flow	The control flow is incorrectly implemented.
	Exception Handling	Does not have proper exception handling.
	Input	Input handling or validation is incorrect or ignored.
	Output	Output displays incorrectly.
	Processing	Data processing such as evaluation of expressions and equations is incorrect.
	Other Wrong Functionality Implementation	Any other *semantic* bug that does not meet the design requirement.

TABLE IV: Interpretation of Kappa values.

Kappa Value	Interpretation
< 0	poor agreement
[0.01, 0.20]	slight agreement
[0.21, 0.40]	fair agreement
[0.41, 0.60]	moderate agreement
[0.61, 0.80]	substantial agreement
[0.81, 1.00]	almost perfect agreement

er two Ethereum projects, ethereum/mist and ethereum/cpp-ethereum, and encounter new bug types. Thus we create two new categories *GUI* and *build* as described in TABLE II. Then we look into the bug reports in the AugurProject/augur and steemit/steem projects. We further create two new categories *compatibility* and *hard fork*.

Step 2: Labeling. The first author and a graduate student independently label 1,108 bug reports of the eight blockchain systems. We use Fleiss Kappa [20] to measure the agreement between the two labelers. The interpretation of Kappa values is shown in TABLE IV. The overall Kappa value between the two labelers on all bug reports is 0.65, which indicates substantial agreement between the labelers. After completing the manual labeling process, the two labelers discussed their disagreements to reach a common decision. While many bug reports are well-described, some with cannot be sorted into sets due to insufficient detail or uncertain root causes. We mark such bug reports as the *unknown*. Meanwhile, we identify the bug reports of *duplicate*, *reopen* and *not-a-bug* types. Those bug reports are excluded from our classification. Note that a *duplicate* or *reopen* bug report is often described by comments or marked by tags, and referred to the related bug reports.

III. Bug Categories

A. Overview

This section answers RQ1. We totally identify 946 unique bugs from the 1,108 bug reports. The overall distribution of bugs based on the 10 categories is shown in TABLE V.

We find that most of the bugs are labeled as *semantic* (67.23%). The number is lower than 70.1% - 87.0% reported in a previous study [14]. Programmers could easily introduce *semantic* bugs due to the lack of a thorough understanding of the system. Additionally, since *semantic* bugs are application-specific, it is hard to automatically detect *semantic* bugs. In order to understand what causes *semantic* bugs, we further break down the *semantic* bugs into subcategories as shown in Table III. 23.9% *semantic* bugs are caused by *missing cases*. *Processing* bugs also account for a large portion of *semantic* bugs. The results are consistent with the previous study [21].

TABLE V: Bug types.

Type	Count	Percentage
Semantic	636	67.23%
Missing Cases	*152*	*16.07%*
Processing	*133*	*14.06%*
Other Wrong Implementations	*80*	*8.46%*
Exception Handling	*70*	*7.40%*
Missing Features	*58*	*6.13%*
Wrong Control Flow	*48*	*5.07%*
Corner Cases	*44*	*4.65%*
Output	*35*	*3.70%*
Input	*16*	*1.69%*
Environment and Configuration	108	11.42%
GUI	66	6.98%
Concurrency	42	4.44%
Build	39	4.12%
Security	18	1.90%
Memory	15	1.59%
Performance	14	1.48%
Compatibility	7	0.74%
Hard Fork	1	0.11%

Environment and configuration bugs have the second highest number of occurrence (11.42%). This might be the case that blockchain systems are often used within various environments by numerous end users. Different environments may reside on various hardware and operating systems with various versions of libraries installed. Even if same versions of libraries are installed, end users could have customized configurations. The bugs in libraries, library version mismatching, as well as incorrect configurations, would lead to *environment and configuration* bugs in blockchain systems.

In the following sections, for each bug type, we would also present representative bug samples. To do so, we first manually extract keywords from the title, body, and discussion of bug reports, pull requests, and commit logs. Next, for each bug type, we group the bug reports with similar keywords together. Finally we randomly choose a bug report from each group.

Of the 1,108 closed bug reports from the studied systems, 946 unique bugs could be classified into 10 categories. Semantic *bugs are the dominant runtime bug type. 23.9%* semantic *bugs are caused by* missing cases. Environment and configuration *bugs have the second highest number of occurrence.*

B. Semantic Bugs

Semantic bugs correspond to inconsistencies with the requirements or the programmers' intention. We find that *semantic* bugs are the dominant bug types in studied projects. The finding is in line with previous work [14] in open source software. The representative *semantic* bug samples include:

1) Missing Cases:

- **ethereum/go-ethereum #2519** Previously it was assumed that when-even type `[]interface{}` was given that the interface was empty. The abigen rightfully assumed that interface slices which already have pre-allocated variable sets to be assigned. This PR fixes that by checking that the given `[]interface{}` is larger than zero and assigns each value using the generic set function (this function

has also been moved to abi/reflect.go) and checks whether the assignment was possible.
- **ethereum/go-ethereum #2255** After some investigations I found that SetReadDeadLine is a cause of these errors https://github.com/ethereum/go-ethereum/blob/develop/rpc/http.go#L115 it seems deadline prolonged only for OPTIONS request and GET & POST requests fall into timeout.
- **ethereum/cpp-ethereum #2156** The fallback function is not part of the external gas estimation. Added fallback function to gas estimation and fixed mix gas estimation.
- **AugurProject/augur #477** Market link doesn't work. load single market first if on market/report detail page.

2) Processing:

- **ethereum/go-ethereum #2194** The default gas price that was being used for transaction wasn't properly using the GPO.
- **bitcoin/bitcoin #6186** The use of x where 8 does not divide x was broken, due to a bit-order issue The use of e.g. 1.2.3.4/24 where the netmasked bits in the network are not 0 was broken. Fix this by explicitly normalizing the network according to the bitmask.
- **AugurProject/augur #710** ExtraInfo field not set for markets created through UI. Fixed extraInfo / detailsText field name; updated currentPeriod and currentPeriodProgress calculations; augur.js@1.8.12.
- **ripple/ripple-lib #204** When subscribing to orderbooks with 'model' event, the last offer created is not pushed correctly to the _offers array. In particular it misses some fields like: 'quality', 'bboknode', 'flags' , 'ledgerentrytype' and others.

3) Exception Handling:

- **ethereum/go-ethereum #869** cmd/geth, cmd/utils: improve interrupt handling. The new strategy for interrupts is to handle them explicitly.
- **bitcoin/bitcoin #6702** GetTempPath has unchecked error conditions. GetTempPath can fail, but its callers do not (all?) check for this failure.
- **steemit/steem #76** Steemd loss sync after a websocket client disconnected. Properly catch errors on async connection and release.
- **AugurProject/augur #718** Transactions Sub-System does properly fail on generate order book failure response. Market created successfully, but during order book generation, an error was returned (below), which ultimately stalls the generate order book method. The UI should reflect this error state, but it looks as though an onSuccess of case.

4) Missing Features:

- **ethereum/go-ethereum #1037** th_estimateGas requires to: ... to: is unknown in case you're creating a contract. Would it be possible to make this optional? I understand the RPC spec has been frozen, but do you think it can be changed to allow "to:" to be optional at this stage? Seems like a fairly minor alteration, but would make the function much more useful.
- **bitcoin/bitcoin #321** Wallet rescan doesn't work properly. rescan seems to work on linux but doesn't (always) work on windows.
- **steemit/steem #274** Deleted comments not updated in feeds.
- **AugurProject/augur #564** The UI isn't claiming a user's winnings after a market is closed atm. It should, and then should update realized profit and loss

5) Wrong Control Flow:

- **ethereum/go-ethereum #1037** There used to be a separate handler for the CLI and GUI versions of the protocol. Unfortunately it seems to have been "misplaced".
- **bitcoin/bitcoin #2178** Remove IsFromMe() call from bool CTxMemPool::accept().
- **ripple/ripple-lib #73** Seed.get_key method does not use account_id. Seed should use the account_id to determine which key to return.
- **AugurProject/augur #177** Update network info on block arrival.

6) Corner Cases:

- **ethereum/go-ethereum #1549** If a reply timeout fired (even just nanoseconds) before the deadline of the next pending reply, the timer was not rescheduled. The timer would've been rescheduled anyway once the next packet was sent, but there were cases where no next packet could ever be sent due to the locking issue above.
- **bitcoin/bitcoin #1902** Flood of retarget messages, if internal miner is used. I'm guessing this occurs (a) only on testnet, and (b) only at a difficulty retarget boundary.
- **AugurProject/augur #677** After some tracing looks to be related to the lack of trade volume (seems to be a corner case) with all prices being returned as 0 and thus causing the price percent change to be shown as 0.00

7) Output:

- **ethereum/go-ethereum #1438** geth log reports unsigned instead of signed transaction hash.
- **ethereum/cpp-ethereum #2217** Alethzero on OS X doesn't display new transactions / closed by the submitter.
- **AugurProject/augur #504** Creation Date sort backwards. Just created 2 new markets and they show up at the top of (oldest first).
- **dogecoin/dogecoin #282** Item "Show transaction of Dogechain" not translated at all. missing translation field.

8) Input:

- **ethereum/go-ethereum #41** Number fields are limited in characters Changed validators to regexp validators IntValidator limits to 32bit int (JavaScript limitation) and therefore the input fields are limited in length.
- **ethereum/cpp-ethereum #2473** Solidity: Exponential notation for constants. Literally "1e20" can't be converted to cpp_int directly, that's why it crashes. If we allow this, we might need to evaluate to cpp_float first and then convert to cpp_int. Check whether a literal is a valid literal before using it.
- **AugurProject/augur #387** Creating markets with negative liquidity. It allowed me to create a market with -20000 initial liquidity.

C. Environment and Configuration Bugs

Environment and configuration bugs correspond to the bugs that lie in third-party libraries, underlying operating systems, or non-code parts (e.g. configuration files). The representative *environment and configuration* bug samples include:

- **ethereum/go-ethereum #1818** Just returning the current path is working for me. Hope it will be a easy fix. It is definitely the `golang` bug as this bug report https://bugzilla.redhat.com/show_bug.cgi?id=1135152 says.
- **bitcoin/bitcoin #3274** You didn't give a –with-boost option to point to the one you compiled, so it's finding the ones on your system (which are presumably borked, which is why you're building your own). Use something like ./configure –with-boost=/usr/local, where /usr/local is the prefix where boost was installed.
- **dogecoin/dogecoin #277** Closing this for now as it's likely either a problem with Berkley DB or caused by a hardware error or similar.
- **ethereum/cpp-ethereum #1982** We can't do anything for cards with less than 1G of memory. Actually cards with 2G of memory is the absolute minimum. This is how the PoW algorithms works. Closing this issue.
- **AugurProject/augur #225** The underlying issue turned out to be a bug in js-ipfs-api. I made an issue for this on the js-ipfs-api Github (ipfs/js-ipfs-api#212) and walled off the bug in AugurProject/ramble@27b6cb7 / 09b1ab3 so that the site will now load on iOS. The IPFS-based features (comments and metadata) will not work until this bug is fixed, but in the meantime the rest of the site should work now.

D. GUI Bugs

GUI bugs correspond to incorrect display of graphic user interface, e.g., incorrect font, text alignment, and button size. We find no *GUI* bugs in project ethereum/go-ethereum, ethereum/cpp-ethereum, and ripple/ripple-lib because they do not have graphical user interface. The representative *GUI* bug samples include:

- **bitcoin/bitcoin #243** In several areas, text appears to be too large to fit into designated areas. I suspect this could be because I have Windows set to use 120 DPI rather than the default 96.
- **dogecoin/dogecoin #511** In the 1.7 Qt client splashscreen text overlaps the logo.
- **ethereum/mist #828** Make new window loader screen frameless. Remove the buttons from the loading new window screen.

E. Concurrency Bugs

Concurrency bugs are synchronization issues among the concurrent threads or processes in concurrent programs [19], including race conditions and deadlocks. The representative *concurrency* bug samples include:

- **ethereum/cpp-ethereum #1159** Address returned by eth_transact on contract creation is not guaranteed to be the address that the contract will actually get, if multiple clients are creating transactions at the same time, making this value useless. eth_setDefault can not be used reliably, because it can be changed any time by any client.
- **bitcoin/bitcoin #453** Deadlock in key generation due to CCryptoKeyStore. I'll be commiting lock order inconsistency detection code tomorrow.
- **AugurProject/augur #340** This is actually intentional. What's happening is that if 2 transactions are created by the UI at roughly the same time, they can be assigned identical nonces. Added a mutual exclusion lock to the ethrpc rawTxs object in AugurProject/augur.js@ae577f4, which should force nonces to always be unique.

F. Build Bugs

Build bugs occur in build processes, e.g., compilation and link errors. We find no *build* bugs in project ethereum/go-ethereum, ripple/ripple-lib, steemit/steem, and ethereum/mist. The representative *build* bug samples include:

- **ethereum/cpp-ethereum #1591** TARGET_PLATFORM is linux even on windows. fixed #1591, cmake TARGET_PLATFORM.
- **bitcoin/bitcoin #879** bitcoin-qt FTBFS : undefined reference to symbol 'shm_unlink@@GLIBC_2.2.5'. We must link to -lrt on Linux, as previously we were using it indirectly through some other lib and that's no longer allowed.
- **ethereum/mist #798** Unable to build wallet for all platforms. Building –platform all on linux will result in broken OSX package. Running –platform darwin seperately will build successfully.
- **AugurProject/augur #713** Running tests when AURC is npm linked fails. Import straight from AURC/src (quick fix).

G. Security Bugs

Security bugs correspond to vulnerabilities that allow attackers to reduce a system's information assurance. The representative *security* bug samples include:

- **ethereum/go-ethereum #1352** SEC-52 block header timestamp int64 overflow. A malicious miner can set a timestamp value that will overflow a int64 into a value that is valid according to consensus rules. This breaks consensus as two different block headers (hashes) would be considered the same by the Go consensus validations.
- **bitcoin/bitcoin #2838** Timing leak in RPC authentication. Mitigate Timing Attacks On Basic RPC Authorization Eliminates the possibility of timing attacks by changing the way the two passwords are compared. See http://rdist.root.org/2010/01/07/timing-independent-array-comparison/ for reference.
- **dogecoin/dogecoin #280** As requested from the poolop channel im pasting the trace of 1.5 dying with a buffer overflow.
- **AugurProject/augur #649** Multiple vulns found by security audit performed by Rapid7. Security / TLS/SSL Server is enabling the BEAST attack ==> ssl-cve-2011-3389-beast.

H. Memory Bugs

Memory bugs usually occur in systems written in memory-unsafe programming languages (e.g. C and C++). Because those programming languages support unsafe pointer operations such as arbitrary pointer arithmetic, casting, and deallocation. The representative *memory* bug samples include:

- **ethereum/cpp-ethereum #888** Crash on exit. Session no longer holds a shared ptr to Node. Please reopen if similar crash (deallocating deallocated Peer) occurs.
- **bitcoin/bitcoin #6963** Hard crash when validateaddress RPC endpoint is requested. Fix a null pointer dereference in validateaddress with -disablewallet.
- **ethereum/go-ethereum #/2605** [SWARM] crash due to out of bound string index. panic: runtime error: slice bounds out of range.

I. Performance Bugs

Performance bugs correspond to the bugs that make system perform its operations slowly or inefficiently. The representative *performance* bug samples include:

- **ethereum/go-ethereum #1334** High disk I/O during statedb write/read.
- **bitcoin/bitcoin #889** Bitcoin-qt using 100% cpu on Mac. boost::interprocess is busy-waiting (at low priority), because it is emulating Posix functionality.
- **ethereum/mist #265** I'm running Mist on Windows 10. I have been trying to download the blockchain since two days ago and the process is too slow and cumbersome.
- **AugurProject/augur #630** There are 3 calls to the toLocaleString method in formatNumber (src/utils/format-number.js). This method is (weirdly) a notorious resource sink and should not be used. I recorded a CPU profile in Chrome (loaded 1100 markets, then stopped it), which revealed that 73.16% of CPU time used by formatNumber.

J. Compatibility Bugs

Compatibility bugs correspond to the bugs that make a system fail to run on a particular CPU architecture, operating system, or Web browser. We find that all of the 7 *compatibility* bugs exist in the AugurProject/augur project.The representative *compatibility* bug samples include:

- **AugurProject/augur #491** Gray screen / browser issues. Also got someone saying it didn't work in IE. Should probably just use the UI on one of those sites that tests dozens of browsers / versions and then see what's going wrong.
- **AugurProject/augur #117** UI doesn't load on the best web browser of all time (Internet Explorer). I only have access to Windows 10 and Edge 20.1 currently and it work splendidly. Can you do a run through on any version of IE you have again? If you do have an issue, please note the platform and browser version.
- **AugurProject/augur #98** On the Javascript console in Firefox (Iceweasel 38.1.0) I get a vast number of "too much recursion" messages.

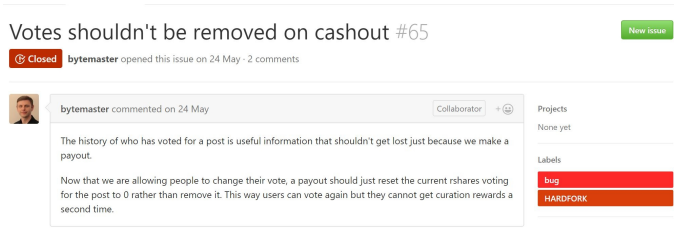


Fig. 1: Example of *hard fork* bug.

K. Hard Fork Bugs

Hard fork bugs are peculiar bug types in blockchain systems. A hard fork is a change to the blockchain protocol that makes previously invalid blocks/transactions valid, or vice versa. *Hard fork* bugs have the least occurrence in our studied systems. Only one *hard fork* bug appears in the steemit/steem project as shown in Fig. 1. The old behavior is that "users can vote again and can get curation rewards again after paid out"; the new behavior is that "users can vote again but cannot get curation rewards again". The change of system behavior causes a change to the consensus in blockchain protocol.

IV. Bug Distribution

A. Across Projects

In order to answer RQ2, we analyze the frequency distributions of bug types of the eight blockchain projects. The frequency distribution over distinct categories of each project is shown in Fig. 2. We can observe similar frequency distributions of bug types across the eight projects. For instance, *semantic* bugs account for the majority in most studied projects.

To check if the differences in the frequency distribution of bug types are statistically significant, we further employ the Wilcoxon signed-rank test [22] at 95% significance level on 10 paired categories. In the 28 Wilcoxon signed-rank tests of the eight projects, all the p-values are greater than 0.05 (min: 0.12, max: 1.00, median: 0.70). This means that the differences in the frequency distribution of bug types across projects are not significant.

The frequency distributions of bug types share similar trends across studied projects.

B. Across Programming languages

In order to answer RQ3, we aggregate eight blockchain systems based on their primary languages. We analyze the frequency distributions of bug types of the three primary languages, i.e., C++, Go, and JavaScript. The bug distribution over distinct categories of each language is shown in Fig. 3. We can observe similar distributions of bug types across the three languages.

To check if the differences in the frequency distributions of bug types of the three languages are statistically significant, we further employ the Wilcoxon signed-rank test [22] at 95% significance level on 10 paired categories. In the 3 Wilcoxon signed-rank tests of the eight projects, all the p-values are greater than 0.05 (min: 0.6356, max: 0.9219, median: 0.9057).

TABLE VI: Bug fixing durations in terms of days.

Type	Min	Max	Mean	Median
Semantic	0.0111	1902.3743	109.8578	16.1337
Env and Conf	0.0167	925.2250	75.4209	20.3125
GUI	0.0188	796.0917	123.6400	46.7799
Concurrency	0.0083	958.1396	154.4956	36.5653
Build	0.0007	782.8500	99.4609	21.8403
Security	0.0361	623.7465	128.3290	**117.1035**
Memory	1.7438	664.0861	90.3282	28.5625
Performance	0.3410	1049.3361	**280.7240**	89.8604
Compatibility	1.1563	83.2861	32.8864	29.4236
Hard Fork	2.7597	2.7597	2.7597	2.7597

This means the differences in the distributions of bug type frequency across programming languages are not significant.

The frequency distributions of bug types share similar trends across programming languages.

V. Bug Fixing Duration

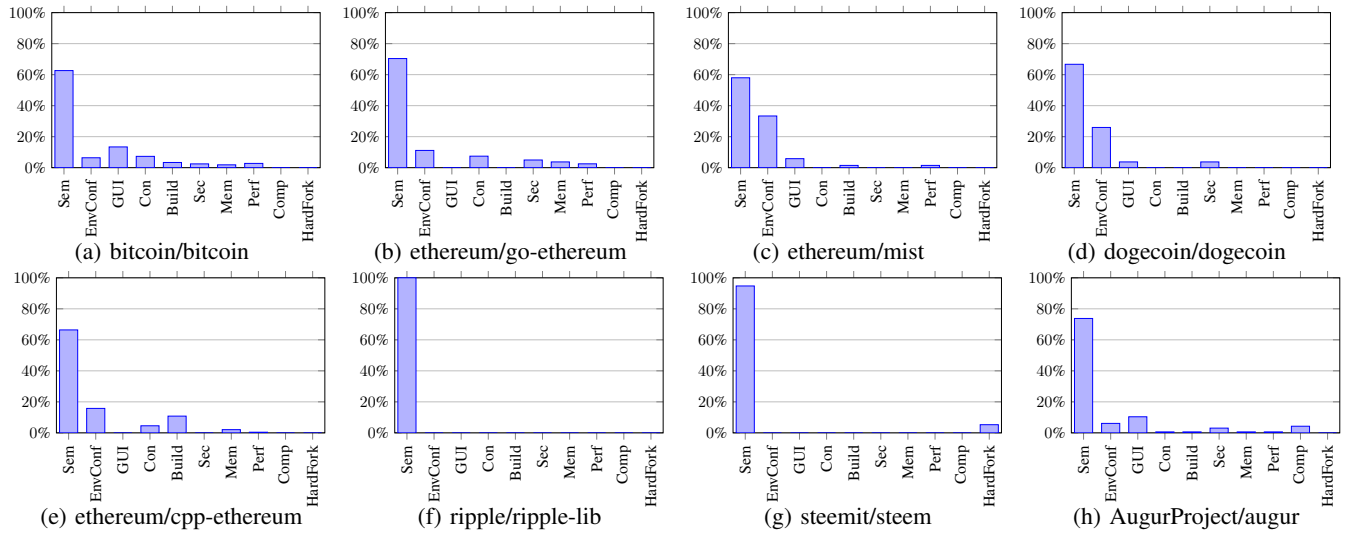
In order to answer RQ4, we investigate the relationship between bug categories and bug fixing time. We measure bug fixing time by the number of days that passed until a bug report is closed and not re-opened (at least until the time we crawled the repository in Nov 2016). TABLE VI shows the minimum, maximum, mean, and median numbers of days that have elapsed before bugs of various categories are fixed.

We notice that the minimum period for all bug categories is just a few minutes between `create` time when a bug is submitted and `close` time when a bug is fixed. Most of these bugs are submitted by project developers. In some cases, the developers may have noticed the bugs earlier; they just submit these bugs right after finding solutions to fix the bugs; they promptly commit related code changes to close the bugs. This finding is in line with the observation in previous work [13, 23]. In other cases, the developers may collect fixed bug reports from external data sources; they just submit and close the bug reports promptly.

The maximum bug fixing time can take a few months or years. Four bug categories with the highest maximum bug fixing time are *semantic*, *performance*, *concurrency*, and *environment and configuration* bugs, whose maximum bug fixing time is more than 900 days. In terms of mean bug fixing time, *performance*, *concurrency*, *security* and *GUI* bugs take the longest time to be closed (more than 120 days).

TABLE VII gives further breakdowns of durations into a month, a year, and more than a year for distinct bug categories. We find that 55.39% bugs are fixed within a month, 34.78% bugs are fixed within a year, and 9.83% bugs are fixed after one year. We manually investigate the bugs that were fixed after many years. Most of those bugs are of *semantic* category. Due to the small sample sizes of bugs of distinct categories, our results may not be statistically significant. It would be future work to collect more relevant bug samples for further analysis.

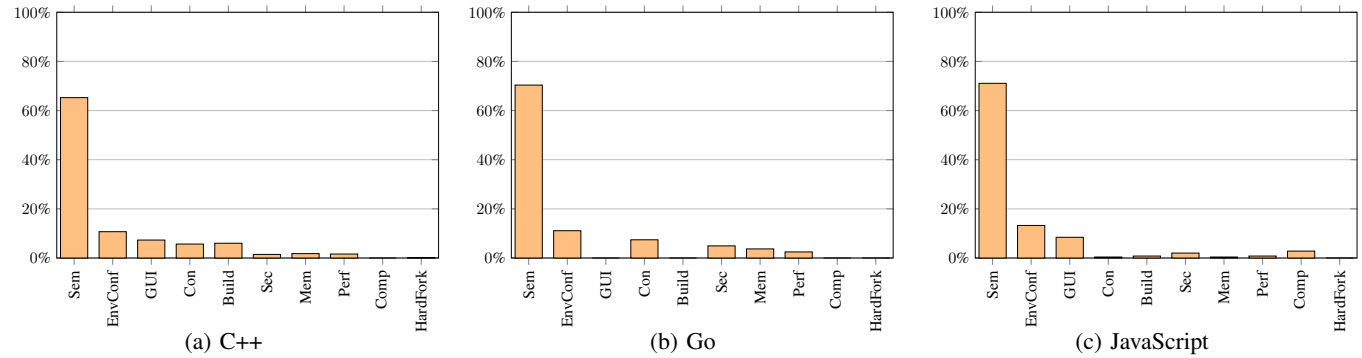
In terms of median bug fixing time, security *bugs take the longest time to be fixed. 35.71%* performance *bugs are fixed in more than one year; in terms of average bug fixing time,* performance *bugs take the longest time to be fixed.*



(a) bitcoin/bitcoin (b) ethereum/go-ethereum (c) ethereum/mist (d) dogecoin/dogecoin

(e) ethereum/cpp-ethereum (f) ripple/ripple-lib (g) steemit/steem (h) AugurProject/augur

Fig. 2: Distribution of bug types for each of the blockchain systems.



(a) C++ (b) Go (c) JavaScript

Fig. 3: Distribution of bug types for each of programming languages.

VI. Discussion

A. Implications

***Semantic* bugs are the dominant runtime bugs.** TABLE V shows that *semantic* bugs represent the majority of the studied bugs, accounting for 67.23% in the studied systems. This number is lower than the 70.1% - 87% reported in Tan et al.'s work [14]. This is because our investigated bugs include non-runtime bug type, i.e., *build* bugs, and non-code bug type, i.e., *configuration and environment* bugs. Specifications could help with preventing and detecting *semantic* bugs. Various approaches have been proposed to mine specifications from program executions [24, 25], and individual versions [26–30] or version histories [31] of source code. Unfortunately, specification mining approaches suffer from high false positive rates. Future studies could be conducted to reduce false positive rates of specification mining approaches. In addition, there is no proof that existing specification mining approaches are practical for blockchain systems. Thus it would be beneficial to understand if existing specification mining tools are capable of detecting *semantic* bugs in blockchain systems.

Among *semantic* bugs, *missing cases* bugs account for 23.9%, which is higher than 7.1% - 23.3% reported in the previous study [14]. *Missing cases* bugs, also known as *neglected condition* bugs, have long been known but hard-to-find software defects [32]. Some *missing cases* bugs can be prevented by the use of requirement elicitation and analysis techniques. Some other *missing cases* bugs involve design or implementation issues that do not correspond directly to requirements. Previous studies discover those bugs by either careful code inspection [33] or specification-based testing [34]. Thus improving design and specification would make a significant difference on the studied blockchain systems.

However, if *missing cases* are neglected in the requirement specification and design documents of a system, those bugs will be difficult to detect. Engler et al. [26] have shown that many software bugs, including aforementioned *missing cases* bugs, can use compiler extensions to match *rule templates* against a code base. Those rule templates are derived from knowledge of typical programming bugs. Various approaches have been developed to mine programming rules [32, 35] to identify *missing cases* bugs. Chang et al. [32] apply frequent subgraph mining on C code to mine condition rules. The scalability of their approach may be limited by the underlying graph mining algorithm. Thummalapenta and Xie [35] use alternative pattern mining on Java code and detect neglected conditions. Their approach may not be precise and cannot identify equivalent conditions (e.g., considering the conditions $a > 0$ and $a \geq 1$ as different). To effectively detect *missing*

cases bugs in blockchain systems, future work could be developing rule mining tools for Go and JavaScript languages, leveraging more precise static analysis, and designing scalable graph mining algorithms.

Exception handling bugs comprise around 11% of the *semantic* bugs. Exception handling code is crucial for a robust system. Incorrect or missing exception handling in security-sensitive code often causes severe security vulnerabilities (e.g., CVE-2014-0092, CVE-2015-0208, CVE-2015-0288, CVE-2015-0285, and CVE-2015-0292) [36]. In addition, systems are expected to handle exception conditions and take necessary recovery actions such as releasing resources. Failing to release resource may cause performance degradation and lead to other critical issues [37]. Thus automated detection and fixing tools for *exception handling* bugs would significantly improve robustness of blockchain systems.

***Environment and configuration* bugs are one of the major types.** TABLE V shows that *environment and configuration* bugs account for 11.42% of the reported bugs in the studied systems. *Configuration* bugs are one of the major causes for the downtime of large-scale enterprise systems [38]. Previous studies have proposed ideas to predict, detect, diagnose, and fix *configuration* bugs [39–43]. In our studied systems, many *configuration* bugs are due to wrong compiler options or wrong configuration of external libraries. The aforementioned research directions could benefit from our bug characteristics study of blockchain systems. Moreover, understanding the characteristics of *configuration* bugs in blockchain systems may help developers to better design configuration logic and requirements, and could thereby reduce the likelihood of *configuration* mistakes by users.

On the other hand, many *environment* bugs are caused by issues in external libraries. For example, bug #1818 in the ethereum/go-ethereum project is caused by a bug in Go language. Libraries have a significant impact on the stability of studied systems. Therefore, software engineering of blockchain systems should not neglect this important factor. Further studies could be conducted on usage analysis, architecture analysis and software quality assessment [44] of third-party libraries.

***Memory* bugs only account for a small portion.** TABLE V shows that *memory* bugs account for a small fraction (1.59%) of studied bugs. This percentage is much lower than the 11.8% - 16.3% reported in previous work [14]. One possible reason for this reduction could be the adoption of programming languages with managed memory type (i.e., JavaScript and Go) [15], as well as the use of more advanced debugging tools during the development process in recent years [14]. The causes include NULL pointer dereference, use-after-free, memory allocation, memory corruption, heap overflow, and double deallocation. Most of *memory* bugs result in a crash. Therefore, future studies could be conducted to understand if existing *memory* bug detection tools are capable of addressing issues in blockchain systems (e.g., improving the capability of *memory* bug detection tools, designing *memory* bug detection tools for Go programming language, and reducing false positives).

TABLE VII: Numbers of bugs fixed within various durations.

Type	Duration	Count	Proportion
Semantic	within a month	373	58.65%
	winthin a year	199	31.29%
	more than a year	64	10.06%
Env and Conf	within a month	59	54.63%
	winthin a year	44	40.74%
	more than a year	5	4.63%
GUI	within a month	28	42.42%
	winthin a year	32	48.48%
	more than a year	6	9.09%
Concurrency	within a month	19	45.24%
	winthin a year	16	38.10%
	more than a year	7	16.67%
Build	within a month	23	58.97%
	winthin a year	12	30.77%
	more than a year	4	10.26%
Security	within a month	5	27.78%
	winthin a year	12	66.67%
	more than a year	1	5.56%
Memory	within a month	9	60.00%
	winthin a year	5	33.33%
	more than a year	1	6.67%
Performance	within a month	3	21.43%
	winthin a year	6	42.86%
	more than a year	5	**35.71%**
Compatibility	within a month	4	57.14%
	winthin a year	3	42.86%
	more than a year	0	0.00%
Hard Fork	within a month	1	100.00%
	winthin a year	0	0.00%
	more than a year	0	0.00%

***Security* bugs take the longest median time to be fixed.** We notice that *security* bugs get fixed slower than non-security bugs. The finding is similar to that of Bhattacharya et al.'s study [45]. Some *security* bugs are generic and cross-projects, i.e., buffer overflow vulnerabilities. We believe blockchain systems would benefit from adopting automatic vulnerability detection tools for buffer overflow vulnerabilities [46, 47]. In addition, application-specific *security* bugs exist in studied blockchain systems, e.g., timing attacks, application-layer denial of service, and missing security checks. Those *security* bugs receive little attention [48]. Further research could be conducted to design detection tools for application-specific *security* bugs in blockchain systems.

35.71% *performance* bugs are fixed in more than one year; *performance* bugs take the longest average time to be fixed. Note that 35.71% of *performance* bugs take more than one year to fix (see TABLE VII). The percentage is much higher than that of non-performance bugs. The average time necessary to fix *performance* bugs is about 280 days (see TABLE VI), which is longer than that of non-performance bugs. These numbers indicate that *performance* bugs are probably more difficult to fix than non-performance bugs. The finding is consistent with the previous study [49]. Current effort on helping developers fix bugs focuses on non-performance bugs [50–53]. Thus more research is needed on automatic repair tools of *performance* bugs.

B. Threats to Validity

Construct validity. We rely on the tag "bug" from the GitHub issue repositories to identify closed bug reports. Thus we may miss some bug reports that are not tagged with "bug" due to ignorance of project contributors. In addition, some "closed"

bugs cannot be sorted into sets due to insufficient detail or uncertain root causes. We omit such bug reports and do not categorize them.

Internal validity. Our card sorting of bug reports is subjected to interpreter's bias. We minimize the subjectivity in manual inspection through double verification: each bug report is examined at least twice by two different people independently. If they disagree, they would discuss and reach a consensus.

External validity. Similar to any characteristics study, our findings should be considered together with our methodology. The eight studied systems are open-source projects written in C++, Go, and JavaScript. Thus our results may not generalize to commercial blockchain systems or systems written in other programming languages.

VII. Related Work

In this section, we briefly review related studies. We first review some previous empirical studies on bugs. Next, we describe studies on bug categorization.

Empirical Study on Bugs. Prior work on bug empirical studies has examined the bug reports of both open source and proprietary software [8, 11, 13, 17, 54–58]. Chou et al. investigate bugs in Linux and OpenBSD kernels [8]. A similar study is performed a decade later by Palix et al. [57]. Maji et al. study defects in mobile operating systems, including Android and Symbian [11]. Sahoo et al. examine reported bugs in server software to facilitate bug diagnosis [58]. Li et al. analyze bugs in Mozilla and Apache Web server [17]. They categorize bugs in three dimensions - root cause, impact, and software component. Seaman et al. investigate bugs in NASA projects and categorize bugs depending on where these defects occur [55]. Thung et al. analyze bugs in three machine learning systems - Apache Mahout, Lucene, and OpenNLP [13]. Different from these studies, we analyze the bugs that appear in open source blockchain systems.

Other work focuses on specific bug categories. Zaman et al. analyze security and performance bugs in Firefox [12] and understand the difference between the two bug categories. They answer questions such as how fast bugs are fixed, who fix bugs, and what characteristics the bug fixes have. Ozment and Schechter [59] measure the rate at which code is introduced and the rate at which security vulnerabilities are reported in the code base of OpenBSD operating system. Then they determine whether OpenBSD's security increases over time. Massacci et al. [60] study reported security vulnerabilities in six major versions of Mozilla Firefox. Neuhaus and Zimmermann [61] study the security vulnerability reports in Common Vulnerability and Exposures (CVE) to find the trend of security vulnerabilities. Lu et al. conduct a detailed empirical study on concurrency bugs in MySQL, Apache Web server, Mozilla, and OpenOffice [62]. They analyze the root causes and types of concurrency bugs, and the number of threads and variables involved in the bugs. Tan et al. [14] study the concurrency bugs in the Linux kernel, Mozilla and Apache. They find that the Linux kernel has more concurrency bugs than the other two non-OS software. In contrast to these studies, we analyze a wide variety of bugs in blockchain systems and analyze how these bugs affect blockchain systems.

Bug Categorization. Previous studies have developed various bug taxonomies for different objectives [7, 10, 14, 21, 55, 63–76]. Some of the classifications are system-specific, the others are generic thus can become the basis for a more specific classification. Vijayaraghavan and Kaner summarize bug taxonomies and their objectives in [74]. Chillarege et al. [21] analyze the symptoms and causes of the defects, and categorize the defects into 5 types in terms of their causes. Later Chillarege et al. [7] present the most widely used classification Orthogonal Defect Classification (ODC). They extend the 5 types in their previous work to 8 types. A number of researchers use ODC as a starting point and develop new taxonomies for specific purposes. For instance, based on ODC framework, Leszak et al. [76] analyze root causes and classify defects by relating them with their underlying causes; Freimut et al. [75] build and validate a defect categorization scheme. In addition, Ma et al. [10, 72, 73] conduct a series of studies on categorization of Web applications faults. Nikora et al. [71] develop a new fault classification, which differs from others in that "it does not seek to identify the root cause of the fault. Rather, it is based on the types of changes made to the software to repair the faults" [70]. A number of taxonomies have been developed specifically for security concerns [67–69]). Seaman et al. [55] aggregate historical defect data using different categorization schemes to guide future development. We use card sorting to identify bug groups in blockchain systems, and give each group a name by referring to [14].

VIII. Conclusion and Future Work

This paper studies the bug characteristics in eight open source blockchain systems. We manually examine 1,108 bug reports in blockchain systems and leverage card sorting to categorize bugs. We further investigate the frequency distribution of bug types across projects and programming languages, and relationship between types and bug fixing time.

In order to reduce the manual effort in categorizing bugs, we plan to use text mining and machine learning techniques to automatically classify hundreds of thousands of bugs. We would like to investigate more blockchain systems and bugs. We also plan to investigate the approaches that could help developers to avoid *semantic* bugs, and the effectiveness of existing tools on *environment and configuration* bugs. Furthermore, we would like to investigate how the distribution of bug types evolves over time as the projects become more stable, and examine bug characteristics by considering bug severity levels.

Acknowledgment

We thank Mengfan Zhu for classifying some bug reports and early discussion. We also thank the anonymous reviewers for their insightful comments. This research is supported by NSFC Program (No. 61602403 and 61572426), and National Key Technology R&D Program of the Ministry of Science and Technology of China (No. 2015BAH17F01).

REFERENCES

[1] CoinMarketCap, "Crypto-Currency Market Capitalizations," https://coinmarketcap.com, retrieved Nov. 2016.

[2] Coindesk, "Bitcoin venture capital," http://www.coindesk.com/bitcoin-venture-capital/, retrieved Nov. 2016.

[3] "The Ashley Madison hack...in 2 minutes," http://money.cnn.com/2015/08/24/technology/ashley-madison-hack-in-2-minutes/, retrieved Nov. 2016.

[4] "Colored Coins," http://coloredcoins.org/, retrieved Nov. 2016.

[5] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 839–858.

[6] "Ethereum white paper," https://github.com/ethereum/wiki/wiki/White-Paper/, retrieved Nov. 2016.

[7] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal defect classification-a concept for in-process measurements," *IEEE Transactions on software Engineering*, vol. 18, no. 11, pp. 943–956, 1992.

[8] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '01. New York, NY, USA: ACM, 2001, pp. 73–88.

[9] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 329–339.

[10] L. Ma and J. Tian, "Web error classification and analysis for reliability improvement," *Journal of Systems and Software*, vol. 80, no. 6, pp. 795–804, 2007.

[11] A. K. Maji, K. Hao, S. Sultana, and S. Bagchi, "Characterizing failures in mobile oses: A case study with android and symbian," in *2010 IEEE 21st International Symposium on Software Reliability Engineering*. IEEE, 2010, pp. 249–258.

[12] S. Zaman, B. Adams, and A. E. Hassan, "Security versus performance bugs: a case study on firefox," in *Proceedings of the 8th working conference on mining software repositories*. ACM, 2011, pp. 93–102.

[13] F. Thung, S. Wang, D. Lo, and L. Jiang, "An empirical study of bugs in machine learning systems," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, 2012, pp. 271–280.

[14] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1665–1705, 2014.

[15] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 155–165.

[16] GitHub, "Linguist," https://github.com/github/linguist, retrieved Nov. 2016.

[17] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now?: an empirical study of bug characteristics in modern open source software," in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. ACM, 2006, pp. 25–33.

[18] D. Spencer, *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[19] S. Lu, "Understanding, detecting and exposing concurrency bugs," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 2008, aAI3347439.

[20] J. L. Fleiss, "Measuring nominal scale agreement among many raters." *Psychological bulletin*, vol. 76, no. 5, p. 378, 1971.

[21] R. Chillarege, W.-L. Kao, and R. G. Condit, "Defect type and its impact on the growth curve," in *Proceedings of the 13th International Conference on Software Engineering*, ser. ICSE '91. Los Alamitos, CA, USA: IEEE Computer Society Press, 1991, pp. 246–255.

[22] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[23] A. Lamkanfi and S. Demeyer, "Filtering bug reports for fix-time analysis," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, 2012, pp. 379–384.

[24] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '02. New York, NY, USA: ACM, 2002, pp. 4–16.

[25] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: mining temporal api rules from imperfect traces," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 282–291.

[26] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '01. New York, NY, USA: ACM, 2001, pp. 57–72.

[27] Z. Li and Y. Zhou, "Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 306–315.

[28] M. Acharya, T. Xie, and J. Xu, "Mining interface specifications for generating checkable robustness properties," in *Proceedings of the 17th International Symposium on Software Reliability Engineering*, ser. ISSRE '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 311–320.

[29] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Path-sensitive inference of function precedence protocols," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 240–250.

[30] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007, pp. 35–44.

[31] B. Livshits and T. Zimmermann, "Dynamine: Finding common error patterns by mining software revision histories," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 296–305.

[32] R.-Y. Chang, A. Podgurski, and J. Yang, "Discovering neglected conditions in software by mining dependence graphs," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 579–596, 2008.

[33] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 38, no. 2/3, p. 258, 1999.

[34] D. R. Kuhn, "Fault classes and error detection capability of specification-based testing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 8, no. 4, pp. 411–424, 1999.

[35] S. Thummalapenta and T. Xie, "Alattin: Mining alternative patterns for detecting neglected conditions," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 283–294.

[36] S. Jana, Y. Kang, S. Roth, and B. Ray, "Automatically detecting error handling bugs using error specifications," in *USENIX Security Symposium (USENIX Security)*, 2016.

[37] S. Thummalapenta and T. Xie, "Mining exception-handling rules as sequence association rules," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 496–506.

[38] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" in *USENIX*

symposium on internet technologies and systems, vol. 67. Seattle, WA, 2003.

[39] X. Xia, D. Lo, W. Qiu, X. Wang, and B. Zhou, "Automated configuration bug report prediction using text mining," in *Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual*. IEEE, 2014, pp. 107–116.

[40] B. Xu, D. Lo, X. Xia, A. Sureka, and S. Li, "Efspredictor: Predicting configuration bugs with ensemble feature selection," in *Software Engineering Conference (APSEC), 2015 Asia-Pacific*. IEEE, 2015, pp. 206–213.

[41] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang, "Automatic misconfiguration troubleshooting with peerpressure." in *OSDI*, vol. 4, 2004, pp. 245–257.

[42] S. Zhang and M. D. Ernst, "Automated diagnosis of software configuration errors," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 312–321.

[43] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do not blame users for misconfigurations," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 244–259.

[44] V. Bauer, L. Heinemann, and F. Deissenboeck, "A structured approach to assess third-party library usage," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 483–492.

[45] P. Bhattacharya, L. Ulanova, I. Neamtiu, and S. C. Koduru, "An empirical analysis of bug reports and bug fixing in open source android apps," in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. IEEE, 2013, pp. 133–143.

[46] J. Viega, J.-T. Bloch, Y. Kohno, and G. McGraw, "Its4: A static vulnerability scanner for c and c++ code," in *Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference*. IEEE, 2000, pp. 257–267.

[47] D. A. Wheeler, "Flawfinder," https://www.dwheeler.com/flawfinder/, retrieved Nov. 2016.

[48] J. P. Near and D. Jackson, "Finding security bugs in web applications using a catalog of access control patterns," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 947–958.

[49] A. Nistor, T. Jiang, and L. Tan, "Discovering, reporting, and fixing performance bugs," in *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*. IEEE, 2013, pp. 237–246.

[50] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 389–400.

[51] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 3–13.

[52] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu, "Automated concurrency-bug fixing." in *OSDI*, vol. 12, 2012, pp. 221–236.

[53] C. Liu, J. Yang, L. Tan, and M. Hafiz, "R2fix: Automatically generating bug fixes from bug reports," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 282–291.

[54] X. Zhao, X. Xia, P. S. Kochhar, D. Lo, and S. Li, "An empirical study of bugs in build process," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 2014, pp. 1187–1189.

[55] C. B. Seaman, F. Shull, M. Regardie, D. Elbert, R. L. Feldmann, Y. Guo, and S. Godfrey, "Defect categorization: making use of a decade of widely varying historical data," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 2008, pp. 149–157.

[56] X. Xia, X. Zhou, D. Lo, and X. Zhao, "An empirical study of bugs in software build systems," in *Quality Software (QSIC), 2013 13th International Conference on*. IEEE, 2013, pp. 200–203.

[57] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, "Faults in linux: Ten years later," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 305–318.

[58] S. K. Sahoo, J. Criswell, and V. Adve, "An empirical study of reported bugs in server software with implications for automated bug diagnosis," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1. IEEE, 2010, pp. 485–494.

[59] A. Ozment and S. E. Schechter, "Milk or wine: does software security improve with age?" in *Usenix Security*, 2006.

[60] F. Massacci, S. Neuhaus, and V. H. Nguyen, "After-life vulnerabilities: a study on firefox evolution, its vulnerabilities, and fixes," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2011, pp. 195–208.

[61] S. Neuhaus and T. Zimmermann, "Security trend analysis with cve topic models," in *2010 IEEE 21st International Symposium on Software Reliability Engineering*. IEEE, 2010, pp. 111–120.

[62] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 329–339.

[63] X. Xia, D. Lo, E. Shihab, X. Wang, and X. Yang, "Elblocker: Predicting blocking bugs with ensemble imbalance learning," *Information and Software Technology*, vol. 61, pp. 93–106, 2015.

[64] X. Xia, D. Lo, E. Shihab, X. Wang, and B. Zhou, "Automatic, high accuracy prediction of reopened bugs," *Automated Software Engineering*, vol. 22, no. 1, pp. 75–109, 2015.

[65] X.-L. Yang, D. Lo, X. Xia, Q. Huang, and J.-L. Sun, "High-impact bug report identification with imbalanced learning strategies," *J. Comput. Sci. & Technol*, vol. 32, no. 1, 2017.

[66] X. Xia, D. Lo, X. Wang, and B. Zhou, "Automatic defect categorization based on fault triggering conditions," in *Engineering of Complex Computer Systems (ICECCS), 2014 19th International Conference on*. IEEE, 2014, pp. 39–48.

[67] T. Aslam, I. Krsul, and E. H. Spafford, "Use of taxonomy of security faults," Purdue University, Department of Computer Science, Tech. Rep., 1996.

[68] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi, "A taxonomy of computer program security flaws," *ACM Computing Surveys (CSUR)*, vol. 26, no. 3, pp. 211–254, 1994.

[69] S. Weber, P. A. Karger, and A. Paradkar, "A software flaw taxonomy: aiming tools at security," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–7, 2005.

[70] A. P. Nikora and J. C. Munson, "Software evolution and the fault process," in *Proc. Twenty Third Annual Softw. Eng. Workshop, NASA/Goddard Space Flight Center*, 1998.

[71] A. P. Nikora, "Software system defect content prediction from development process and product characteristics," Ph.D. dissertation, University of Southern California, 1998.

[72] L. Ma and J. Tian, "Analyzing errors and referral pairs to characterize common problems and improve web reliability," in *International Conference on Web Engineering*. Springer, 2003, pp. 314–323.

[73] Z. Li and J. Tian, "Analyzing web logs to identify common errors and improve web reliability," in *Proceedings of the IADIS International Conference on e-Society, Lisbon, Portugal*, 2003, pp. 235–242.

[74] G. Vijayaraghavan and C. Kaner, "Bug taxonomies: Use them to generate better tests," *STAR EAST*, 2003.

[75] B. Freimut, C. Denger, and M. Ketterer, "An industrial case study of implementing and validating defect classification for process improvement and quality management," in *11th IEEE International Software Metrics Symposium (METRICS'05)*. IEEE, 2005, pp. 10–pp.

[76] M. Leszak, D. E. Perry, and D. Stoll, "Classification and evaluation of defects in a project retrospective," *Journal of Systems and Software*, vol. 61, no. 3, pp. 173–187, 2002.