# Smart Contract Development: Challenges and Opportunities

Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach Dinh Le, Xin Xia, Yang Feng, Zhenyu Chen, Baowen Xu

**Abstract**—Smart contract, a term which was originally coined to refer to the automation of legal contracts in general, has recently seen much interest due to the advent of blockchain technology. Recently, the term is popularly used to refer to low-level code scripts running on a blockchain platform. Our study focuses exclusively on this subset of smart contracts. Such smart contracts have increasingly been gaining ground, finding numerous important applications (e.g., crowdfunding) in the real world. Despite the increasing popularity, smart contract development still remains somewhat a mystery to many developers largely due to its special design and applications. Are there any differences between smart contract development and traditional software development? What kind of challenges are faced by developers during smart contract development? Questions like these are important but have not been explored by researchers yet. In this paper, we performed an exploratory study to understand the current state and potential challenges developers are facing in developing smart contracts on blockchains, with a focus on Ethereum (the most popular public blockchain platform for smart contracts). Toward this end, we conducted this study in two phases. In the first phase, we conducted semi-structured interviews with 20 developers from GitHub and industry professionals who are working on smart contracts. In the second phase, we performed a survey on 232 practitioners to validate the findings from the interviews. Our interview and survey results revealed several major challenges developers are facing during smart contract development: (1) there is no effective way to guarantee the security of smart contract code; (2) existing tools for development are still very basic; (3) the programming languages and the virtual machines still have a number of limitations; (4) performance problems are hard to handle under resource constrained running environment; and (5) online resources (including advanced/updated documents and community support) are still limited. Our study suggests several directions that researchers and practitioners can work on to help improve developers' experience on developing high-quality smart contracts.

**Index Terms**—Smart Contract, Challenges, Empirical Study, Blockchain

## 1 Introduction

Since the release of Bitcoin in 2009 [105], decentralized cryptocurrencies have gained considerable attention and adoption [2]. For instance, till February 2018, the numbers of coins and tokens hosted on the coinmarketcap[1] were 896 and 649, respectively. A cryptocurrency is administrated not by a central authority, but by automated consensus among networked users. The users in the cryptocurrency network run a consensus protocol to maintain and secure a public and append-only ledger of transactions, i.e., blockchain. In recent years, the potential of blockchain technology has been exploited beyond cryptocurrencies, among which a promising use of blockchain is *smart contract*.

The term "smart contract" was originally coined to refer to the automation of legal contracts in general [140]. The term was (and is still) used to refer to a legal contract which or at least parts of which is capable of being expressed and implemented in software [66]. The advent of blockchain technology has recently brought much interest on smart contracts. Today, the term is popularly used to refer to as code scripts that run synchronously on multiple nodes of a distributed ledger (e.g., a blockchain) [30]. In this paper, we mainly focus on the latter, more specific definition of smart contracts, i.e., low-level code scripts running on blockchains.

As a program running on a blockchain, a smart contract can be correctly executed by a network of mutually distrusting nodes without the need of an external trusted authority. The self-executing nature of smart contracts provides a tremendous opportunity for use in many fields that rely on data to drive transactions [139]. In the beginning of 2018, more than 10% of the jobs advertised on Guru[2] (one of the biggest freelancer sites) were related to smart contracts and blockchains [56]. Currently, more and more developers are devoting themselves to developing smart contracts in various domains, e.g., finance, game, and notary [11]. The number of smart contracts deployed on Ethereum[3] (the most popular public blockchain for running smart contracts with market capitalization exceeding $80 billions) has also sharply increased to more than 2 million in March 2018[4].

The emergence of smart contracts brings about a growing and widespread interest in the research community. More and more researchers are taking smart contracts as study targets [21],


- *Weiqin Zou, Yang Feng, Zhenyu Chen, and Baowen Xu are with State Key Laboratory for Novel Software Technology, Nanjing University, China. E-mail: wqzou@smail.nju.edu.cn, charles.fy0708@gmail.com, zychen@nju.edu.cn, bwxu@nju.edu.cn*
- *David Lo is with the School of Information Systems, Singapore Management University, Singapore. E-mail: davidlo@smu.edu.sg*
- *Pavneet Singh Kochhar is a software engineer in Microsoft, Canada. E-mail: kochharps.2012@phdis.smu.edu.sg*
- *Xuan-Bach Dinh Le is with School of Computing and Information Systems, University of Melbourne, Australia. E-mail: bach.le@unimelb.edu.au*
- *Xin Xia is with Faculty of Information Technology, Monash University, Australia. E-mail: xin.xia@monash.edu*
- *Zhenyu Chen is the corresponding author.*


[1]http://coinmarketcap.com

[2]http://www.guru.com/

[3]http://www.ethereum.org/

[4]https://etherscan.io/accounts/c. Last Access: March 2018

[90], [69]. A growing number of papers have been published in events such as ACM/IEEE International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)[5], and International Workshop on Blockchain Oriented Software Engineering (IWBOSE)[6], as well as some tracks at conferences such as ACM Conference on Computer and Communications Security (CCS) and International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)[7][122], etc.

Despite the increasing popularity of smart contract, the potential challenges that developers are facing when developing smart contracts have not yet been clearly explored. Without understanding these challenges, practitioners and researchers may spend much efforts developing techniques and tools that are not appreciated by developers and thus are underused in practice.

To help advance research in smart contract development, we conducted an empirical study to explore the work practice and potential challenges faced by developers during smart contract development on blockchains, with a focus on Ethereum. We followed a mixed-method approach that is a combination of interviews (qualitative) and survey (quantitative). Specifically, we first interviewed 20 developers with different backgrounds and expertise. During interviews, we asked participants about their normal work practices and relevant challenges faced during different stages (e.g., coding, testing, debugging) of smart contract development. Then we used open card sorting [135] to analyze the interview results. The resulting categories produced by open card sorting were grouped into six groups, i.e., security, debugging, programming language, Ethereum virtual machine, gas, and online learning resources & community support. After that, we performed a validation survey with 232 developers to confirm various insights from the interviews, including challenges, best practices, and desired improvements.

Based on the interviews and survey, we found that developers cared a lot about code security but did not have effective ways to prove the correctness, reliability and security of their code; meanwhile, the lack of powerful tools especially step-through/interactive debuggers often made it painful to program smart contracts. Furthermore, as mentioned by developers, the current programming languages and virtual machines that were specifically designed for smart contracts still had a number of limitations (e.g., lack of general purpose libraries and limited support for debugging). These limitations often adversely affected their project development experience, especially for complex projects. Another big challenge for developers was performance issues - they were interested in tools and resources that could help them write efficient smart contracts that consume fewer resources on the blockchain. Besides, lack of advanced/updated documents and delay in responses from the online community also had an impact on smart contract development.

The major contributions of our study are as follows:

- To the best of our knowledge, this is the first in-depth study that explores practitioners' perceptions on current state of smart contract development and challenges ahead through interviews and survey.
- We perform an analysis of the qualitative and quantitative data and highlight actionable insights and implications that developers, tool builders, and researchers can use to improve developer experience during smart contract development.

[5]http://conferences.computer.org/icse-w/2018/#!/toc/28
[6]http://saner.unimol.it/blockchainOrientedSoftwareEngineering
[7]http://www.isola-conference.org/isola2018/tracks.html

The remaining parts of this paper are structured as follows: In Section 2, we provide background materials on smart contracts. In Section 3, we present our empirical study methodology in detail. The findings of our study are presented in Section 4. Section 5 presents some potential research directions based on our findings. Section 6 discusses the threats to validity of our study. The last two sections present the related work and summarize our study.

## 2 Background

**Blockchain.** A blockchain in its very simple form is a chain of records called blocks, in which blocks are linked and secured using cryptography. Each block is characterized by some transaction data, a time stamp, and the hash value of its previous block. Blockchain can be considered as a public ledger where each block contains records of some transactions. The blockchain is not stored in a single location but on a network of nodes, where each network node has a copy of this blockchain. This means all the records are public and easily verifiable to all network nodes, which makes it very expensive for a node to modify any data in the blockchain. Once a block is appended to the blockchain, it is extremely hard to modify the block's transactions without achieving consensus of all nodes. All these features are by design and based on peer-to-peer consensus protocol [105]. The blockchain technology allows two untrusted parties to make transactions securely without the participation of a trusted third party. This makes blockchain suitable for record keeping tasks such as storage of ownership rights of musical work, financial transactions, etc. Cryptocurrencies which are based on blockchain have attracted considerable attention lately [90]. An emerging area of blockchain technology is smart contract.

**Smart Contract.** The term "smart contract" was coined by Nick Szabo in the mid 1990s [140]. He suggested translating the clauses of a contract into code and embedded them into software or hardware to make them self-execute, in order to minimise contracting cost between transacting parties and to avoid accidental exceptions or malicious actions during contract performance.

Currently people in different disciplines used the term "smart contract" in different ways. Some referred "smart contract" as a legal contract which (or at least elements of which) could be represented by software. While some others took "smart contract" as code scripts which are designed to execute certain tasks once pre-defined conditions are met; these scripts typically (although not necessarily) run on distributed ledgers (e.g., blockchains) [27], [136]. Clack et al. proposed a definition of smart contract which is broad enough to cover the breath of above-mentioned definitions. They defined a smart contract as "an automatable and enforceable agreement. Automatable by computer, although some parts may require human input and control. Enforceable either by legal enforcement of rights and obligations or via tamper-proof execution of computer code." [29].

In this paper, we mainly focus on low-level code scripts running on blockchains. As a program running on a blockchain, a smart contract can facilitate a contract between two parties without relying on a trusted third party. Technically speaking, a smart contract is a program that contains both data (e.g., account balance) and executable code. Smart contract can be stored in the blockchain, and can be automatically executed when certain pre-condition is met. After each execution of the smart contract, its state can be updated on the blockchain [152].

**Smart Contracts Running on Corda.** Corda is an open-source permissioned blockchain platform that is explicitly designed to account for the highly regulated environment of the financial service industry [92]. Within Corda, each node has a certificate that maps their network identity to a real-world legal identity. The communication between Corda nodes is point-to-point and the transaction history is fully encrypted and private to only necessary parties [17]. Smart contracts running on Corda are allowed to consist of both code and legal prose [146]. The associated legal prose could be referred back to traditional legal systems in case of legal disputes in smart contract performance. A smart contract in Corda has three key elements, namely executable code, state objects, and commands [119]. The executable code mainly validates the changes to state objects in transactions. State objects are data that record the existence, content and current state of an agreement between two or more parties, and work as input or output of transactions. Commands are additional data that are included within transactions. They mainly describe what is going on and tell the executable code the way to verify a transaction. All smart contracts could be programmed in Kotlin or Java and could be compiled into Java Virtual Machine (JVM) bytecode.

**On-Chain and Off-Chain Smart Contracts.** Due to the nature of blockchain technology, smart contracts deployed on blockchains (i.e., on-chain smart contracts) generally need to be executed and validated by each node, with all relevant transactions being visible to the entire blockchain network. This reduces the privacy of smart contracts. Further, for smart contracts especially those with complex computation, the transaction cost may be high (e.g., users need to pay gas fee for transactions on Ethereum) and the validation of relevant transactions may take a long time (due to replicated execution of smart contracts among nodes). As an alternative solution towards these problems, the idea of "off-chain" smart contract has been proposed [41], [83]. Off-chain smart contracts are executed outside of the blockchain. Unlike on-chain smart contracts, an off-chain smart contract only needs to be signed and executed by interested participants. As proposed, an off-chain smart contract is generally designed to encapsulate functions involving high-cost computation or private information about the participants; while an on-chain smart contract is suggested to conduct some low-cost and non-sensitive tasks. To preserve the properties and benefits of a blockchain, in practice, the results of off-chain smart contracts would be for example logged on-chain [41]. In case of any disagreement on the execution results of an off-chain smart contract, an on-chain smart contract may be used to fork the off-chain smart contract and execute it on blockchain to solve the dispute [83].

**Blockchain Platforms for Smart Contracts.** Blockchains can be divided into public and non-public categories. Public blockchain platforms allow any user to join the network while non-public blockchain platforms allow only permitted users to join. Examples of public blockchains are Ethereum, and NEO[8]. Some examples of non-public blockchains are Fabric[9] and Quorum[10]. Different blockchain platforms provide different support for smart contracts. Some (e.g., Bitcoin) may only allow users to use a simple scripting language to develop smart contracts with simple logic; while some platforms, such as Ethereum, support much more advanced programming languages for writing smart contracts [127].

[8]http://neo.org/
[9]http://www.hyperledger.org/
[10]http://www.jpmorgan.com/global/Quorum

**Ethereum.** Since its release in July, 2015, Ethereum has grown to become the most popular blockchain platform for smart contracts [21]. Ethereum provides a decentralised Turing-complete machine, namely the Ethereum Virtual Machine (EVM), to execute scripts using an international network of public compute nodes [19]. On Ethereum, people can use programming languages, e.g., Solidity[11] and Vyper[12], to develop complex smart contract applications. All smart contracts written in high-level languages would be compiled to the same format, i.e., Ethereum bytecode, and be executed by the EVM. Ethereum also has its own cryptocurrenty, namely *Ether*. Ether can be transferred between accounts and used to compensate participants who mine blocks for computations performed [19].

**Gas.** Ethereum adopts an internal pricing mechanism, i.e., *gas* for all transactions running on it [19]. Gas is a measure of how much computing resource a transaction would cost. People need to pay gas fee (in Ethers) for each transaction they make; and a transaction would fail if it runs out of gas. If users want to have their transactions mined by miners faster, they can choose to increase the gas price. By using the gas mechanism, Ethereum is able to better allocate resources and mitigate spam on the network.

**Trusted Execution Environment (TEE).** To preserve the confidentiality or privacy of smart contracts, some efforts are being made to integrate trusted execution environments (TEEs) with blockchains [26], [70], [46]. A TEE is a secure area of a main processor which ensures sensitive data to be stored, processed, and protected in an isolated environment called an *enclave*. Data inside an enclave could only be accessed by code residing in the same enclave. Within an enclave, both the code and data are protected by hardware enforced access control policies. The operating system and other applications are not able to tamper with or eavesdrop on the state of any application running inside the enclave unless the hardware is breached. There have been several realizations of TEE, including Intel SGX (the most prominent TEE technology today) [5], TrustZone [7], etc. By using a TEE, one does not need to trust the host (which runs the blockchain code) of the enclave.

## 3 METHODOLOGY

Figure 1 shows the overview of our methodology design. On the whole, our study includes two parts: a series of interviews with 20 expert developers to get insights into smart contract development and a follow-up survey to validate the findings of the interviews. We describe the details how we conduct the interview and survey below.

### 3.1 Interview

**Protocol.** In our study, we conducted semi-structured interviews [133]. Specifically, we began each interview with an introduction, a short explanation of our research, and some demographic questions about the interviewee. Next, we used some open questions to guide the discussion – some of them are listed in Table 1. The full list of open questions can be found at https://github.com/SurfGitHub/smartcontractStudy/blob/master/interview_questions.pdf. These open questions probed our interviewees about their views on major differences between smart contract development and traditional software development and their impacts, challenges involved in performing various smart

[11]http://github.com/ethereum/solidity
[12]http://github.com/ethereum/vyper

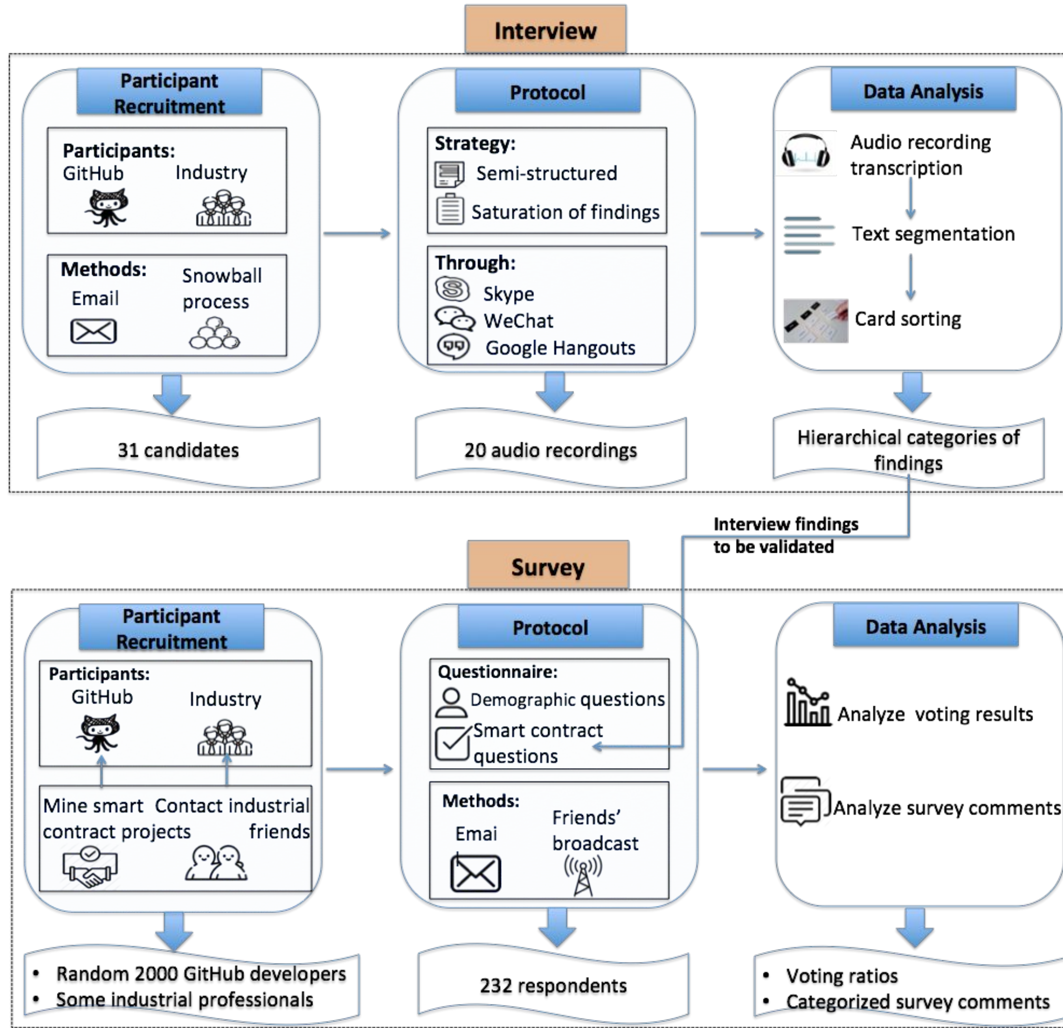


Fig. 1. Overview of methodology design.

contract development activities, etc. Since the interview was semi-structured, we also asked follow-up questions to dig deeper into our interview participant's viewpoints when appropriate. At the end of the interview, we asked the interviewee to provide any other important information that we may have missed during the interview.

We totally had 31 interviewee candidates (more details in Participant Recruitment below). During interviews, we followed the methodology employed in [6], [134] to decide when to stop interviewing, i.e., stopping interviews when saturation of findings was reached. Saturation is a widely-used methodological principle in qualitative research [106], [101], [55]. It is often taken to indicate that further data collection or analysis are unnecessary based on the data that have been collected or analyzed hitherto [124]. More specifically, if the collected data is considered already sufficient and further data collection does not generate new information, then the sampling should not be continued [137].

Taking into account the findings in behavior and brain sciences which claim that "there is substantial variability in experimental results across populations" [58], we made sure to interview participants from different backgrounds (as shown in Table 2) before deciding whether saturation had been reached. During each interview, the authors (who conducted interviews) worked together to ask questions and take notes. Upon finishing an interview, they would compare their notes with previous ones to check whether the interview was bringing any new insights. Finally, we stopped our interviews when we achieved saturation of the findings after we interviewed 20 people.

All interviews were performed remotely via Skype, WeChat, or Google Hangouts, and were audio recorded with the permission of participants. The average and standard deviation of the interview time were 52.34 and 15.74 minutes, respectively. Table 2 shows the basic demographics of the interviewees. According to the table, the interviewees had an average experience of 11.35 years in general software development, and 1.27 years in smart contract development by the time of interviews. Besides, they held various roles, including developers, testers, project managers, architects, designers, CEO/CTO, research assistants, and smart contract trainers. This to a large extent, guarantees the heterogeneity of those 20 persons.

**Participant Recruitment.** We contacted potential participants in multiple ways. We sent emails to smart contract developers on GitHub. We also contacted some developers in well-known com-

TABLE 1
Open Interview Questions (Excerpt).

| ID | Question |
|---|---|
| 1 | What are the main differences between smart contract development and traditional software development? |
| 2 | How do the differences affect your smart contract development? |
| 3 | What practices do you often use to ensure code quality? |
| 4 | What kind of tools do you often use to develop smart contract? Are they useful enough? Why? |
| 5 | What is your way to debug/test smart contract? Any problems encountered? |
| 6 | Do you think current programming languages are good enough? Why? |

panies working on smart contract development such as ConsenSys[13], and OpenZepplin[14]. Then, we expanded the initial group by using a snowball process [47], i.e., adding additional participants recommended by current participants. Finally, 31 persons agreed to participate in our interviews.

**Data Analysis.** For each interview, the audio recording was first transcribed into text. After that, the first author read the transcripts and converted them into separate units each with coherent meaning. Then, we performed card sorting [135] to identify the categories from these units. Card sorting is a commonly used technique that helps to derive categories from data [75], [76]. There are three types of card sorting, namely closed card sorting with predefined categories for data, open card sorting with no predefined categories (i.e., the categories of open card sorting are totally derived from the data), and a hybrid card sorting which combines the previous two types [159]. Considering our study is an exploratory one with categories (i.e., challenges of smart contract development) being unknown in advance, we decided to adopt an open card sorting process to analyze the interview data.

Specifically, during card sorting, after a card was created for each textual unit, the cards were then clustered into meaningful groups, with each group having a topic or a theme. These groups, i.e., low-level subcategories, further evolved into high-level categories. The results of such an open card sorting would let us obtain a hierarchical structure of the categories. Four researchers including two non-authors were involved in the card sorting process. Each card was analyzed and verified by two researchers. Through card sorting, we identified six high-level categories, i.e., security, debugging, gas, programming language, the Ethereum Virtual Machine, and online resources & community support, with each category containing several subcategories (more details in Section 4).

### 3.2 Survey

**Design.** Our survey includes a number of demographic questions and smart contract questions. The demographic questions are mainly designed to understand the background and experience of respondents. Specifically, we created 8 demographic questions that ask respondent's main role (e.g., development, testing, etc.), experience in software engineering and smart contract development, country, highest educational qualification, and the kind of projects and blockchains they mainly work on.

The smart contract questions are designed to validate insights that we got by analyzing the interviewee comments. For each of the six categories (i.e., security, debugging, programming language, Ethereum virtual machine, gas, and online resources &

TABLE 2
Basic information of interviewees. General Exp. and SC Exp. represent a participant's experience in general software development and smart contract development respectively (in years) till the time of interview.

| ID | General Exp. | SC Exp. | Role |
|---|---|---|---|
| P1 | 21 | 0.7 | CEO and Developer |
| P2 | 17 | 0.7 | Architect |
| P3 | 15 | 2.8 | CEO, Core developer of Ethereum |
| P4 | 15 | 2.0 | Developer |
| P5 | 6 | 1.5 | Developer |
| P6 | 4 | 0.6 | Research Assistant |
| P7 | 9 | 1.5 | Project Manager |
| P8 | 6 | 1.5 | Developer |
| P9 | 6 | 0.8 | Developer |
| P10 | 7 | 0.9 | Tester |
| P11 | 18 | 1.8 | Developer |
| P12 | 22 | 2.0 | CEO, Advisor, CTO |
| P13 | 15 | 0.6 | Developer |
| P14 | 15 | 1.6 | Blockchain speaker and trainer, co-founder |
| P15 | 8 | 0.5 | Developer, co-founder |
| P16 | 7 | 1.5 | Developer, Token Sale Advisor |
| P17 | 15 | 0.8 | Project Manager |
| P18 | 4 | 2.8 | Designer |
| P19 | 9 | 1.5 | Developer |
| P20 | 8 | 0.3 | Developer |

community support) that we identified by analyzing our interview responses, we created a set of survey questions. In total, we created 27 questions. For some of them, we asked respondents to rate statements on a Likert scale of 1 to 5 (1 = strongly disagree, and 5 = strongly agree). For some others, we asked respondents to pick one or a few out of a number of options. Two of the questions were open-ended. The full list of our survey questions can be found at http://github.com/SurfGitHub/smartcontractStudy.

**Survey Respondent Recruitment and Statistics.** Our potential survey respondents are developers who contributed to smart contract related projects on GitHub. To identify those projects, we first referred to the topic list[15] of GitHub, and chose three topics that are most related with smart contracts, i.e., Ethereum, Solidity and Truffle[16]. Then we downloaded all the repositories under these three topics. To ensure that we did not miss any popular smart contract projects, we further used GitHub's search API to get a list of projects whose *name, description* or *readme* contain the keyword "smart contract". After that, we selected and manually checked 1,000 smart contract related repositories that have the most stars. For each repository, we obtained the email accounts by analyzing its commit logs. In the end, we had 4,466 distinct email addresses. Out of these, more than 2,590 developers were involved in multiple repositories. Then we randomly selected 2,000 smart contract developers and sent our survey invitations to them through emails. In two weeks time, we got 205 responses, with a response rate of 10.2%. This response rate is similar to those reported by prior studies [94], [118]. Besides GitHub developers, we also asked our friends in industry to help broadcast our survey to their friends and colleagues who may be interested to participate in our survey. With their help, we got another 27 responses from industry. In the end, we had 232 responses from respondents coming from 48 countries.

Among the 232 respondents, 81.9% respondents' main role is development; and 43.1% respondents have advanced degrees (e.g., Master, Ph.D.). The respondents have different expertise in

[13]https://new.consensys.net/
[14]https://openzeppelin.org/
[15]GitHub Topics list can be accessed at https://github.com/topics.
[16]http://truffleframework.com/

general software development and smart contract development: 32.8% of the respondents have >10 years of general software development experience, and 66.8% have >4 years of experience; 13.8% respondents have >2 years of smart contract development experience, and 46.6% have >1 year of experience. We found that 69.4% of the respondents mainly developed smart contracts on public blockchains; and the ratio of developers who mostly spent time on open source projects and closed source projects are 40.1% and 36.2%, respectively.

**Data Analysis.** After we got all the responses from respondents, we adopted different analysis methods for closed-ended and open-ended questions. Specifically, for each closed-ended question, we counted the votes that went to each answer option. Then for each answer option, we calculated its vote ratio by dividing the number of votes for the option over 232 (i.e., the number of all respondents). While for the open-ended questions, we collected all the comments respondents made. Then we removed some comments that were either not meaningful (e.g., "Yes") or not related to our research topic (e.g., "happy to help you with your research"). After that, we tried to place the remaining comments into their corresponding categories obtained from the interview analysis. If a comment involved more than one category (e.g., belonging to both debugging and security), then we split it into separate comments with each assigned to only one category. Whenever we present survey comments, we refer it as (S?), e.g., S5 refers to the fifth survey respondent.

To better understand participants' perspectives of smart contract development, we divided all survey respondents into different demographic groups, and compared their voting results towards various challenges and desired improvements mentioned by interviewees. Following prior studies [89], [80], we considered the following demographic groups:

- Respondents who are developers (Dev)
- Respondents who are testers (Test)
- Respondents who are project managers (PM)
- Respondents with high experience in general software development (>=10.0 years[17]) (seExpH)
- Respondents with low experience in general software development (<=3.0 years[17]) (seExpL)
- Respondents with medium experience in general software development (remaining respondents with more than 3.0 but less than 10.0 years of experience) (seExpM)
- Respondents with high experience in smart contract development (>=2.0 years[18]) (scExpH)
- Respondents with low experience in smart contract development (<=1.0 year[18]) (scExpL)
- Respondents with medium experience in smart contract development (remaining respondents) (scExpM)
- Respondents with advanced degree, e.g., Master, Ph.D. (Adv)
- Respondents without advanced degree (nAdv)
- Respondents who mainly develop on public blockchains (pubBlk)
- Respondents who mainly develop on non-public blockchains (nPubBlk)
- Respondents who develop on both public and non-public blockchains (bothBlk)

For each demographic group, we calculated the number of respondents who said yes or (strongly) agree, as well as the number of respondents who said no or (strongly) disagree, to individual challenges and desired improvements (mentioned by interviewees), respectively. Then following [160], we adopted Fisher's exact test [43] with Bonferroni correction [98] on these numbers to see whether one group tended to vote differently from other group(s). Fisher's exact test is a statistical significance test used in the analysis of contingency tables, which displays the frequency distribution of the variables (i.e., the numbers of yes/(strongly) agree and no/(strongly) disagree votes from each group in our study). It could assess whether the observed difference between two proportions, e.g., the ratios of yes/(strongly) agree votes from two groups, is statistically significant. Bonferroni correction could help to control the family-wise error rate when conducting multiple comparisons. Section 4.7 presents the detailed analysis results.

## 4 FINDINGS

In this section, we first report our findings for each of the six categories that were identified by using open card sorting on interview contents. Each category has several subcategories. For each subcategory, we pick some of the most meaningful comments and highlight some statistics that we derived based on our survey responses to highlight the generalizability of the findings. Then we present the voting results of each demographic group towards those challenges and desired improvements mentioned by interviewees, as well as relevant significance tests over these results. Last, we provide a brief summary of our interview and survey results.

### 4.1 Security

#### *4.1.1* High requirement for code security

Based on our interviews and survey, we found that there was a very high emphasis on ensuring code security for smart contracts. Security concerns bypass all other factors, as highlighted by one survey respondent: *"Contract security concerns and operational security concerns when managing deployed contracts (e.g. key management, contract artifact management) permeate all decisions" (S71).* In our survey, we found that 75.0% respondents agreed to the assertion that smart contract development has a much higher requirement for code security than traditional software development. Based on the reasons highlighted by interviewees, we were able to find three major themes on why there was an increased focus on security in smart contract development:

**Sensitive Nature of Information Handled.** Since smart contracts often control and manage sensitive digital assets (such as virtual currencies, token, digital art files, etc.), people naturally show greater concern for its security, than they do for traditional software. As P20 stated: *"Developer is dealing with money or money flows through code. People would of course have a high requirement on the code security because it controls their assets".*

**Irreversible Transactions.** Unlike traditional software development, users cannot recover any loss they experience while making

[17] The threshold settings of 10 and 3 helped us obtain three expertise groups with roughly equal numbers of respondents -- the numbers of respondents with high experience (>=10 years), medium experience (3-10 years) and low experience (<=3 years) in general software development were 76, 79, and 77, respectively).

[18] As the first platform that supports general smart contract development, Ethereum (released on July 30, 2015) was only about 3.0 years old by the time we conducted our survey. This indicated that even the most experienced developers would generally have no more than 3.0 years of experience in smart contract development. Taking this into account, we chose 2 and 1 as thresholds in determining whether a developer has high experience (>=2 years), medium experience (1-2 years), or low experience (<=1 year) in smart contract development.

transactions on a blockchain based financial system using smart contracts. Since smart contracts run on the blockchain (on which transactions cannot be reverted), if you lose your money, you lose it forever. One developer mentioned: *"Smart contract development is very unforgiving in the sense that you might lose a lot of money and it is impossible to get back. You know, we cannot revert any transactions on blockchain" (P12).*

**Code unmodifiable after deployment.** Code of smart contract cannot be changed after it has been deployed to the blockchain. Unlike traditional software, developers cannot provide a patch to fix a bug. As P9 stated: *"Smart contracts are fundamentally different than regular programming languages due to the blockchain. Once deployed, smart contracts are difficult to change".*

#### *4.1.2* Hard to guarantee security

We found that 71.6% survey respondents agreed that it was difficult to guarantee the security of smart contacts during development. Based on our interviews, we were able to uncover four major aspects of these difficulties.

**Public code access.** As highlighted by one interviewee, the code of smart contract (e.g., on Ethereum) is always publicly accessible. This means that anyone can try to exploit the code, design an attack, and execute it. Also, as smart contracts deal with money, they are always under focus by attackers willing to exploit any loophole. Such reasons place a great burden on smart contract developers who need to secure their code from many potential attacks. One developer mentioned: *"Blockchain environment provides a kind of a unique environment, because all of the code runs on the ecosystem is deployed publicly and is accessible to anyone. Anyone can exploit it and may conduct an attack if they find some security vulnerability within code. You need to think like a hacker and defend your code. It is not easy to anyone" (P3).*

**Flaws in compiler.** Another factor that makes it hard to guarantee code security lies in the compiler itself. Compared with compilers for traditional programming languages, compilers for smart contracts are not mature enough. Many security bugs have been found within smart contract compiler[19]. Also as the compiler is continually evolving, new bugs may be uncovered in future which developers are not even aware of at present. As P4 stated: *"People have discovered a list of security bugs within different compiler versions. I do not know what new bugs we may encounter within the compiler since it still evolves. This is bad I mean, you risk your Solidity code at unseen flaws within future-version compiler".*

Besides, as some survey respondents commented, the compiler changes constantly and does not always have backward compatibility, which makes it hard to ensure the correctness of code especially for a long-running project. *"Due to constant changes on compilers, backward compatibility usually is a problem." (S43) "Solidity changes very rapidly, and complains that code that identifies itself as for an older compiler version uses deprecated features that were current at that version. This means that, for a long-running project, you either have to update your old contracts to new language versions (and have them not correspond to the real, deployed code when you run tests), or you have to ignore a screen full of warnings every time you build." (S69)*

**Lack of best practices for writing safe code.** Many interviewees highlighted the fact that it is harder for them to find coding and security best practices for smart contract development than for traditional software development. As mentioned by several interviewees, there are efforts by organizations such as ConsenSys, to constantly develop and organize some common best practices to help developers in writing safe code. P2 mentioned, *"ConsenSys wrote quite a good guide on smart contact best practices. We would always check its update and tend to adopt them".* However, developers said that such efforts still fall short of meeting the requirements of smart contract application development. One interviewee mentioned, *"ConsenSys or one of the big companies doing smart contracts may have that best practices. I can't find it as an independent developer. When I develop something new, I just cannot find any best practice that help me to make it safe. So I am gonna do with whatever I think is the best" (P13).*

**Lack of tools/techniques to verify code correctness.** In traditional software development, developers could use various tools to help them ensure the quality of their code. Whereas in smart contract development, many interviewees complained that there are no mature tools to verify the correctness of smart contacts. Specifically, they mentioned two kinds of tools that they desired most to better help guarantee code security: code auditing tools and formal verification techniques (49.1% and 42.2% survey respondents listed them as their most desired tools, respectively).

- **Code auditing tools** are the ones which can help developers to discover bugs, security breaches or violations of programming conventions. As P1 stated: *"There are no reliable code auditing tools to help you do a comprehensive analysis of smart contract code. I hope we can have such tools to help us analyze the code, and tell us whether there are some potential bugs, security problems, or convention violations".*
- **Formal verification techniques** ensure the security of code because they are based on mathematical proof. Some interviewees and survey respondents said that they hoped more research work can be done. As one survey respondent stated: *"Mature and robust formal verification tools would be a godsend; code coverage can only get you so far in terms of correctness. Call for formal verification" (S128).*

#### *4.1.3* Current best practices for security

As writing secure code is one of the major focus of developing smart contract applications, we asked developers what steps they followed to ensure security in face of a number of challenges. Interviewees mentioned that testing and code review are their major ways to ensure the correctness of smart contracts, which are discussed in details below.

**Testing.** As P11 stated: *"To ensure the quality of smart contract, I think one best practice is to mostly make heavy use of unit testing".* To better understand the situation of smart contract testing in practice, we asked interviewees what kinds of testing they conducted and what kinds of code coverage they used, then we asked the potential challenges they faced during smart contract testing. We verified their answers in the survey. Our survey results show that 84.9% developers conducted unit testing, 61.6% developers performed integration testing, and 25.4% developers performed performance testing. The most code coverage used by them was function coverage (with 68.1% votes); the statement, branch, and condition coverages were less preferred with 37.1%, 34.1%, and 35.8% developers mentioning that they used them, respectively.

Despite the small/medium size of smart contract program[20], 72.4% survey respondents agreed it was more difficult to test

[19] https://solidity.readthedocs.io/en/develop/bugs.html

[20] After exploring the size of source code from 10000+ smart contracts on Ethereum, we found that more than half smart contract have <300 code lines.

TABLE 3
Major challenges of testing smart contracts.

| Challenge | Votes |
|---|---|
| Difficult to consider all corner cases and scenarios | 69.4% |
| Potential unseen flaws in compilers and virtual machines | 53.4% |
| No mature testing frameworks like other languages | 40.5% |
| Testing needs to be done in an asynchronous way | 31.0% |
| Lack of useful guidance for testing | 28.0% |
| No tool to measure the quality of smart contract test suite | 22.4% |
| Testing consumes gas if tested on testnets or mainnet | 22.4% |

smart contracts than traditional software projects. Table 3 presents the major challenges of testing smart contract rated by survey respondents. The top three challenges are: (1) developers need to consider all corner cases and scenarios; (2) there exist potential unseen flaws in compilers and virtual machines themselves; (3) there are no mature testing frameworks like other languages, e.g., Java.

**Code review.** 84.9% survey respondents agreed that code review is an essential way to ensure the correctness of smart contacts. Our survey statistics do reflect that different kinds of reviews are performed in reality: 83.6% respondents said they would often perform peer code review within team; 26.3% respondents said they would often request help in GitHub for code review; and 27.2% developers said they would often hire third party agency to audit their code.

Meanwhile, compared to traditional software development, some interviewees mentioned that it is more costly to perform code reviews for smart contracts. They mentioned two major challenges of code review that were also verified by our survey results. One challenge is that it is very time consuming to conduct code review (agreed by 66.4% of respondents). One developer commented, *"Within our company, all members of our team participate in the code review. We sit together, read and sometimes discuss code line by line. It is indeed good for improving code quality, but it is too time consuming" (P8).*

The other challenge is that it is very difficult to find qualified developers to find security flaws in smart contracts (agreed by 80.2% respondents). One survey respondent commented that you cannot find people help you unless you pay them: *"It's hard to find another developer to test or even read your smart contract without spending extra funds. I think developers that involved in open-source, should help each other" (S32).*

## 4.2 Debugging

### 4.2.1 Debugging is painful

During our interviews, most participants complained that it is more painful to debug smart contract code compared to traditional software development. In our follow-up survey with developers, 88.8% survey respondents also agreed that it is difficult to debug smart contract applications. In our semi-structured interviews, two main categories of debugging challenges came up, which were also given as answer options to developers during our survey. The categories are briefly described below:

**Lack of powerful interactive debuggers.** As smart contract development is a very recent technology, there is a lack of powerful debuggers in this domain. As one interviewee explained, *"Current debuggers, e.g., Remix, can only provide bytecode level debugging (which requires high skills of developers) and basic interactions, you cannot use it to e.g., visualize the memory state, step through the code line by line and check the current values of variables" (P1).* 69.0% survey respondents also agreed that there is a lack of powerful interactive debuggers, which makes debugging painful and challenging during smart contract development.

**Non-informative error messages.** Some interviewees highlighted the fact that apart from the lack of debugging tools, Solidity (the language used for smart contract development) and EVM (the run time environment used for smart contracts) have a poor support for displaying informative error messages. One interviewee explained, *"Solidity cannot support people to e.g., print error messages in code. Instead, we can only use events or throw exceptions to track the state of the transactions" (P19).* EVM in some cases does not even provide support to display error messages for certain failures such as when a transaction fails. This was highlighted by one of the interviewee who said, *"Sometimes when transactions fail, EVM even cannot throw out the exception" (P2).* In that case, developers totally have no idea what went wrong.

### 4.2.2 Current debugging practices

As highlighted in previous section, there is a lack support for debugging for smart contract development; we were curious to explore if smart contract developers follow certain practices while debugging their code. Based on our interviews and survey results, we summarize the current debugging practices followed by smart contract developers below :

- In our survey, 65.1% respondents said that they use existing debugging tools, e.g., Remix or truffle debugger, to debug buggy code. However, another 65.1% respondents mentioned that they often manually comment out code step-by-step to narrow down buggy code search space.
- 56.5% respondents mentioned that they would often write additional methods/events to check variables and transaction states. This can be attributed to the fact that existing tools do not support checking variable values and transaction states,
- 17.2% respondents of our survey mentioned that they would often request the help of GitHub community or other developers through some forums, e.g., Stack Overflow, when they encounter bugs.

All aforementioned debugging practices, as some interviewees mentioned, are *"very primitive and very inefficient" (P7).* They hope that in the future, the community can develop some powerful debugging tools and can help developers to find *"an easy way to quickly visualize the effects of a smart contract, such as a particular execution, such as showing the call graph for a smart contract in a solidity dependency graph format, and allowing you to highlight a particular section, debug just the execution section. I think things like that would make an enormous difference" (P11).*

## 4.3 Programming Language

### 4.3.1 Limitations of Solidity

Unlike traditional software which are developed in mature general-purpose programming languages (e.g., Java/Python), most smart contracts are developed in specifically-designed programming languages (e.g., Solidity). Through our survey, we found that the programming languages themselves are a major barrier during smart contract development. 39.7% of our survey respondents, agreed that this is one of the top 3 concerns. There are several specific programming languages (e.g., Solidity, Vyper, Bamboo[21]) that can be used for smart contract development. However, as P12 stated: *"Only Solidity is ready for production and used by many*

[21] http://github.com/pirapira/bamboo

*developers, others are still under experiments"*. In practice, even Solidity has issues, as mentioned by some interviewees. Since it has emerged only in the last 3 years, it is still not mature and has many limitations. Based on our survey and interviews, we found that the major limitations of Solidity include:

**Lack of general purpose libraries.** Based on our survey statistics, 56.9% survey respondents said that they often reuse existing libraries for their own development. However, 77.2% respondents agreed that the existing libraries are not enough for smart contract development. Some interviewees and respondents said they need to implement various kinds of libraries (such as string manipulation libraries) by themselves again and again. *"There is a strong need for a well-tested (ideally: verified) standard library for smart contract development. The current state leads to reinventing-the-wheel over and over again for simple things such as string manipulation." (S105)*

**Lack of support for error logging/reporting.** Unlike other traditional programming languages, Solidity does not support direct printing (or logging) of errors, thus developers face a lot of challenges in developing and debugging smart contracts. As one developer opined, *"In terms of the language, I think if a few features such as error reporting is available, that will make a big difference to ease the developments in the future" (P11).*

**Lack of standards/rules.** Several interviewees mentioned that there is a scarcity of standard/rules (e.g., like the ERC20 token standard interface) which can serve the whole development community. P3 mentioned: *"Providing standard interface (such as the ERC20 for token), is even more important than providing general purpose libraries. We lack standards in this field currently"*.

**Lack of safety checks for data types.** Two interviewees mentioned that Solidity does not provide a good check for the safety of data types. P14 stated: *"Solidity does not do well in checking the safety of data types. The compiler does not help you enough. We cannot rely on the compiler to let us know there could be a bug here"*.

**Inconvenient way to call external functions.** Many interviewees mentioned that passing parameters to call external functions is odd, e.g., a developer cannot directly pass his/her own defined structure to the function, instead, he/she should split them and pass them one by one. P17 stated: *"In solidity, struct is only recognized within a smart contract. If Solidity can create a way of packaging the struct in a transportable sort of data structure that would be useful"*.

**Lack of support for memory management.** One interviewee mentioned that Solidity allocates memory in an invisible way that you cannot control. This makes it difficult for developers to develop smart contract in a resource constrained environment. *"Solidity disguises some of the underlying operations of what you're doing more than I would prefer personally, so sometimes you can do things in Solidity that appeared simple but actually resulted in conflicts underlying state changes or more work than you would expect. It does things like allocate memory invisibly and you have no control over that, so in a resource constrained environment sometimes that can be less than ideal." (P11)*

**Constrained number of local variables.** Some interviewees said that Solidity supports a limited number of local variables and to solve this problem, developers need to use more state variables, which affects the efficiency of code. *"If a function uses more than 16 local variables, it cannot be compiled. So you may have to use state variables; but they are slow to read and write as they are stored in the storage rather than in the stack or memory. If you do not want to lose program efficiency, you may have to refactor your code." (P8)*

TABLE 4
Improvements developers would most like to have in Solidity.

| Improvement | Votes |
|---|---|
| More general purpose libraries | 53.0% |
| More powerful error logging/reporting functions | 48.7% |
| More standard interfaces (e.g., ERC20) | 45.7% |
| Better support for security checking of data types | 44.8% |
| More convenient and secure way to call external functions | 35.8% |
| More powerful memory management | 18.1% |
| Loosen the limited number of global and local variables | 13.4% |
| I think Solidity is good enough | 6.5% |
| Others | 5.6% |

#### *4.3.2* Most desired Solidity improvements

To help the community solve the limitations that developers are most concerned about, we asked each survey respondent to select up to 3 improvements that they would like to see in Solidity. Since it is not possible to cover all limitations by interviews, we provide an "Other" text option that allows respondents to fill relevant improvements they would like to have, which did not come out during interviews. Besides, we provide an "I think Solidity is good enough" option for answer completeness.

Table 4 shows the votes. In Table 4, we observe that only 6.5% of the survey respondents agreed that Solidity is good enough. Most developers' concerns are mainly focused on the availability of libraries (including general purpose libraries (with 53.0% votes) and some standard interfaces (with 45.7% votes)), error reporting functions (with 48.7% votes), data type checking (with 44.8% votes), and better way to call external functions (with 35.8% votes).

### 4.4 Ethereum Virtual Machine (EVM)

#### *4.4.1 Limitations of EVM*

Unlike traditional software which run on mature and well-tested virtual machines like JVM and CLR, smart contracts on Ethereum blockchains are executed by a relatively new virtual machine, namely Ethereum Virtual Machine (EVM). Compared to traditional VM like JVM, the current EVM has several limitations. Our survey results show that 35.3% of respondents voted that limitations of the current EVM to be one of the top-3 major challenges that prevent them from effectively developing smart contracts. Four main limitations of EVM mentioned by our interviewees are as follows:

**Limited support for debugging.** When failures happen, developers need help to know where, why, and how their code fails. Unfortunately, the support of debugging features that can provide this needed information is limited in EVM. For example, although EVM supports throwing exceptions, no informative error messages are given to developers, thus giving no clues on what might be the root cause of the problem. As commented by a developer, *"You can only throw exceptions in code. But actually, when your transactions fail in EVM, sometimes even the exceptions cannot be thrown. In that case, we totally have no idea what is going on" (P2).*

**Lack of support of traditional languages.** Popular programming languages (e.g., Rust or Python) are not supported by EVM. EVM instead only supports languages such as Solidity and Vyper, which are newly invented by the smart contract community. Thus, developers' familiarity with popular programming languages may

TABLE 5
Improvements that developers would like to have the most in EVM.

| | |
|---|---|
| Better support for debugging | 65.5% |
| Improve execution speed of byte code | 31.9% |
| Loosen the stack size limit | 27.6% |
| Ability to support traditional languages | 26.7% |
| I think EVM is good enough | 12.5% |
| Others | 9.6% |

not be applicable for those EVM-supported languages, incurring considerable amount of learning cost. *"I am familiar with Python. It is really good. Is it possible for EVM to support python-like language?" (P6)*

**Inefficiency of bytecode execution.** Execution of bytecode in EVM is not speedy due to its design to be single-threaded, according to our interviewees. To mitigate this problem, developers have to find ways to execute bytecode more efficiently by themselves. *"EVM is a single-threaded machine that cannot run transactions in parallel. I mean it is inefficient in executing bytecode. This may be a big problem for people who have a higher requirement on the timely reaction and verification of their transactions. And this further makes developers' life harder." (P4)*

**Limited stack size.** The EVM is a stack machine and all the computations are performed on an area called the stack. The stack has a maximum size of 1024 items with each item having a size of 256 bits. This limited stack size could make it very painful for developers to code their smart contracts. One interviewee said that even a slightly complex smart contract would easily reach the limit of the stack. In this case, a considerable amount of work is required to redesign the code. *"We once developed a relatively complex application, that application could not be compiled only because we have an additional temporary arguments in code...To solve this problem, we had to split one function into several small functions, which makes the code very ugly." (P1)*

#### *4.4.2* Most desired improvements for EVM

Next, we want to identify the most desirable improvements of EVM for which the community should focus more on. Our survey lists a number of EVM's desirable improvements (as shown in Table 5) derived from our initial interviews. From this list, our survey respondents can select the improvements that they desire the most. In addition to the four predefined choices given to respondents, we also provided an "Other" option to allow survey respondents to propose complementary improvements that they wish to have.

Table 5 depicts EVM improvements options given to respondents and their popularity among respondents' choices. The results suggest that better support for debugging is desired the most (65.5% of survey respondents pick this improvement), followed by improvement in execution speed of bytecode (31.9%). The ability to support other programming languages is desired by 26.7% of respondents. Interestingly, although many of our interviewees initially wish EVM to loosen stack size limit, it is desired by only 27.6% of the respondents in the survey. Respondents that are satisfied with current features that EVM offers only made up of 12.5% of total votes.

### 4.5 Gas

Some interviewees mentioned that one significant difference between smart contract development and traditional software development lies in the gas mechanism. The gas mechanism is unique to smart contract development, where the execution of smart contracts would cost gas and users need to pay the gas fee. As a result, developers need to pay special attention to gas consumption during smart contract development. Some interviewees also mentioned some difficulties they encountered in handling gas problems.

#### *4.5.1* Special attention to gas consumption

As mentioned in Section 2, platforms like Ethereum use the gas mechanism to control the executions of smart contracts. Majority of interviewees mentioned that gas consumption deserves special attentions. This is also later validated in our survey – 86.2% of survey respondents declared that they (often) paid attention to gas consumption when developing smart contracts. According to our interviewees, two reasons for why gas consumption is specially important are as follows:

**Gas is money.** One interviewee explained that, on public blockchain platforms like Ethereum, all the resources that a smart contract used would translate into actual direct costs that need to be paid by users in terms of gas. In other words, *"Gas is money for users" (P1)*, thus developers need to be much more conscious on resource consumption. *"Contracts for the Ethereum blockchain have to be executed under very tight constraints. All the resources they used would translate into actual direct costs." (P11)*

**Transaction failure due to insufficient amount of gas.** Some interviewees mentioned that on EVM, if a transaction of smart contract is not given sufficient amount of gas, the transaction might fail. Indeed, our survey results showed that 35.3% of respondents often encountered transaction failures caused by running out of gas. *"You can specify how much gas your transaction is allowed to use; if your transaction run out of gas, it would fail. I often met transaction failures due to insufficient gas for my application." (P6)*

#### *4.5.2 Difficulty in handling gas problems*

In our survey, 63.4% respondents agreed that gas optimization is always painful, especially for complex applications. According to our interviewees, two aspects that contribute to difficulties in performing gas optimization are as follow:

**No gas estimation tool at source code level.** Developers often desire to write and optimize source code rather than bytecode, because it is more intuitive when working at source code level. Unfortunately, there currently exists no gas estimation tool for source code. To optimize their source code with respect to gas consumption, developers thus have to alternatively resort to available gas estimation tools at bytecode level (such as Remix[22]), which may not fully reflect the effect of changes at source code. This approach is hence not intuitive and error-prone, rendering it difficult for developers to perform source code optimization. *"We only have bytecode level dynamic gas estimation tools. What we do right now to optimize code, is to modify the code and run the modified program, and try to compare the gas consumption with the previous program before modified. It is very time consuming to do this actually." (P2)*

A high demand on effective source-code-level gas estimation tools is mentioned by a majority of our interviewees. Such tools, which can directly identify the piece of source code that is most gas costly, would be of tremendous value, according to interviewees. *"We have a bad need in gas estimation tools. Ideally, I hope we can have a tool that does not need to compile your code and can tell you how much gas each source code line costs." (P1)*

[22] http://remix.ethereum.org

**Tradeoff between gas optimization and code readability.** According to our interviewees, optimizing gas without hurting code readability is often a tricky problem. *"If you want to spend less gas, you have to make your code more efficient, so shorter basically, have fewer instructions. But if you have fewer instructions, it tends to make your code less readable as well, so it's a dilemma." (P14)*

### 4.6 Online Resource and Community Support

Some interviewees told us that for traditional software development, we can get a lot of help online when we encounter problems; while for smart contract development, the resource and mentors are scarce because *"smart contract is very new on the blockchain" (P16)*. 22.8% of the survey respondents voted that lacking enough online learning resource and supportive community is one of the top-3 major challenges that prevent them from effectively developing smart contracts.

#### 4.6.1 Online learning resources

After analyzing the interview and survey results, we find developers mainly mentioned three kinds of online learning resources that are missing, i.e., reference code, standardized knowledge, and up-to-date documentations.

**Lack of reference code.** Some interviewees told us that since there are not enough online reference code to reuse, when they build new smart contract applications, they have to build them from scratch. *"So I programmed a lot in python and c++ and javascript, and they all have frameworks, they all have lots of code. The solidity has nothing to test your code on. If I want to do something new, there was nothing on the internet like that, I had to invent it, I feel like from scratch." (P13)*

**Lack of standardized knowledge.** One survey respondent commented that no strict standardized knowledge can guide developers to write better code in an easier way. *"Chaotic non standardized knowledge. i.e., no strict standards (even though there are recommendations - erc). Community intentionally (scam) or unintentionally allow bugs that are later exploited. Basically not mature enough approaches." (S92)*

One interviewee highlighted the importance of coding convention and best practices. *"What's missing is, when you write python code there are code standards, and I didn't find anything like that for Solidity. I don't know what best practices are for code, and so even if I want it to follow them I couldn't." (P13)*

Some interviewees think we should have guides to help developers better test their smart contracts, and as shown in Table 3, 22.4% survey respondents consider the lack of testing guidance as a major testing challenge. *"I do not know what is the best way to do testing, there is no testing guidance that I can follow." (P10)*

**Lack of up-to-date documentations.** Some interviewees mentioned that documentations are often out-of-date due to the quick evolution of relevant tools. Such outdated documentation often make developers feel helpless to make full and correct use of the tools. *"Right now you have documentation about Truffle, about Solidity, about web3, about testrpc. They're all separate, they are all evolving at different speeds, and they are not updated as fast as they could. In the end, developers need to use all those tools together, and yet the documentation is really inconsistent and not always up-to-date." (P14)*

One interviewee suggested that it is necessary to enrich the documentation for some important tools (such as Truffle), e.g., by trying to provide more code examples of some medium and complicated applications. *"I think Truffle would be better if it had more code examples. Truffle, when I used, it had like 'hello world', how to get say 'hello' and had a smart contract, smart token, and that's all that had. There's a lot of things people are building with truffle, but they really had nothing, they didn't have a lot of examples for you to build up the truffle, for me these are really simple examples." (P13)*

#### 4.6.2 Community support

Some interviewees said that although the community support for smart contract development is increasing, the support is still limited. When they encounter some problems or want to ask for some help to e.g., review their code, they cannot easily find relevant developers. One of them mentioned, *"Since the technique is new, the community is still in development. Sometimes you cannot get timely help from the community when you get stuck" (P16).* Another interviewee P13 commented, *"if you go on code review for a javascript or python, you will get lots of people who give you feedback, but in Solidity, you got no feedback. As a hobby developer, we rely on the community to give us feedback and code review, and if you don't have that, we're gonna do whatever we think is right".*

### 4.7 Survey Results

Table 6 lists 28 challenges and desired improvements mentioned by interviewees in the above Sections 4.1 to 4.6. C1 to C6 were six major challenges on the whole of smart contract development. C7 to C17 were challenges developers were facing during different stages (e.g., coding, testing, debugging) of smart contract development. I18 to I28 represented desired improvements of Solidity and EVM, respectively. The last column of Table 6 is the number (ratio) of respondents who voted for the corresponding challenges or desired improvements. For challenges/improvements with "(top-3)", the values represented how many respondents rated them as one of the top-3 challenges or desired improvements. For example, 166 (71.6%) out of 232 respondents rated C1 as one of top-3 challenges (out of six) during their smart contract development.

As the overall voting results of individual challenges or desired improvements have been mentioned in Section 4.1 to 4.6, here we mainly focus on analyzing the voting results of different demographic groups towards these 28 challenges and desired improvements (c.f., Section 3.2 for a description of the methodology that we follow). Table 7 shows the detailed voting results.

From Table 7, we could observe that the voting results varied from demographic groups. For example, for C2, the ratios of scExpM and scExpL were 57.9% and 58.1% while the ratio was only 34.4% for group scExpH. Another example, for C3, the ratios of Dev, Test, and PM were 38.4%, 83.3%, and 47.6%, respectively. To check whether the observed ratio differences are statistically significant, for each challenge/desired improvement, we applied Fisher's exact test with Bonferroni correction on five sets of demographic groups, i.e., groups with different roles (Dev vs. Test vs. PM), groups with different experience in general software development (seExpH vs. seExpM vs. seExpL), groups with different experience in smart contract development (scExpH vs. scExpM vs. scExpL), groups with different education degrees (Adv vs. nAdv), and groups working on different kinds of blockchains (pubBlk vs. nPubBlk vs. bothBlk).

After conducting 392 (14 group pairs × 28 challenges/improvements) Fisher's exact tests with Bonferroni corrections, we found that there were three tests showing that the relevant difference is statistically significant. They are scExpM vs. scExpL on C6 (`p-value=0.002<0.05/3` after Bonferroni correction), pubBlk vs. nPubBlk on I24 (`p-value=0.006<0.05/3` after Bonferroni correction), and Adv vs. nAdv on I28 (`p-value=0.038<0.05`, Bonferroni correction is not needed for the single test). Based on the testing results, we can say with some certainty that:

- Developers with low experience in smart contract development (scExpL) are significantly more likely to rate C6 (limited online learning resource and community help) as a major challenge they are facing during smart contract development, than those with median experience (scExpM) (32.3% vs. 13.2%).
- Developers who mainly worked on non-public blockchains desired I24 (loosening the limited number of global and local variables of Solidity) more, than those mainly working on public blockchains (pubBlk) (37.5% vs. 9.9%).
- Developers without advanced degree (nAdv) desired I28 (EVM's support for traditional programming languages) more than those with an advanced degree (nAdv) (32.1% vs. 19.8%).

### 4.8 Summary of Results

Through the analysis of interview and survey data, we could find that:

- Smart contract has a high requirement for code security. However, developers currently have no effective way to assure code security; some tools like code auditing and formal verification techniques are highly desired. Currently, developers mainly used testing and code reviews to help ensure code correctness.
- Current debugging tools are primitive and inefficient, which makes debugging very painful in practice; more powerful interactive debuggers which provide informative error messages are badly needed.
- Undesirable characteristics of Solidity language (e.g., difficulty in passing data to external functions, limitations in the number of variables), compiler (backward compatibility and reliability issues due to rapidly changing compiler and its unseen flaws) and EVM (e.g., non-informative error messages, limited stack size, inefficient execution due to single-threaded EVM), make it very challenging to program smart contracts effectively and efficiently in practice.
- There is a need for source-code-level gas-estimation and optimization tools that consider code readability.
- There is a lack of best practice, code examples, community support, third-party libraries, and standards for smart contract development.

TABLE 6
28 challenges and desired improvements (mentioned by interviewees) with survey voting results.

| ID | Challenges/Desired improvements | #Votes (Ratios) |
|---|---|---|
| | **Major challenges on the whole** | **(top-3)** |
| C1 | It is hard to guarantee the security of smart contracts. | 166 (71.6%) |
| C2 | There is a lack of powerful tools (e.g., debugger, testing framework). | 127 (54.7%) |
| C3 | Current programming languages have a number of limitations. | 92 (39.7%) |
| C4 | The Ethereum virtual machine that runs smart contracts have a number of limitations. | 82 (35.3%) |
| C5 | It is hard to handle performance problems. | 79 (34.1%) |
| C6 | Online learning resources and community support are limited. | 53 (22.8%) |
| | **Challenges of debugging, gas optimization, and code review** | |
| C7 | It is difficult to debug during smart contract development. | 206 (88.8%) |
| C8 | Doing gas optimization is always painful especially for complex applications.. | 147 (63.4%) |
| C9 | It is hard to find qualified developers to find security flaws in smart contract code. | 186 (80.2%) |
| C10 | Code review of smart contracts is very time consuming. | 154 (66.4%) |
| | **Challenges of testing** | **(top-3)** |
| C11 | Difficult to consider all corner cases and scenarios. | 161 (69.4%) |
| C12 | Potential unseen flaws in compilers and virtual machines. | 124 (53.4%) |
| C13 | No mature testing frameworks like other languages, e.g., Java. | 94 (40.5%) |
| C14 | No tools to measure the quality of smart contract test suite. | 72 (31.0%) |
| C15 | Testing needs to be done in an asynchronous way. | 65 (28.0%) |
| C16 | Testing consume gases if tested on testnets or mainnet. | 52 (22.4%) |
| C17 | Lack of useful guidance for testing, e.g., best practice, tutorials, etc. | 52 (22.4%) |
| | **Desired improvements of Solidity** | **(top-3)** |
| I18 | More general purpose libraries. | 123 (53.0%) |
| I19 | More powerful error logging/reporting functions. | 113 (48.7%) |
| I20 | More standard interfaces. | 106 (45.7%) |
| I21 | Better support for security checking of data types. | 104 (44.8%) |
| I22 | More convenient and secure way to call external functions. | 83 (35.8%) |
| I23 | More powerful memory management. | 42 (18.1%) |
| I24 | Loosen the limited number of global and local variables. | 31 (13.4%) |
| | **Desired improvements of EVM** | |
| I25 | Better support in debugging. | 152 (65.5%) |
| I26 | Improve execution speed of byte code. | 74 (31.9%) |
| I27 | Loosen the stack size limitation. | 64 (27.6%) |
| I28 | Be able to support other traditional languages. | 62 (26.7%) |

## 5 FUTURE DIRECTIONS

### 5.1 Security and Reliability of Smart Contracts

Developers perceive security to be critical to smart contracts. Past reports highlight a wide range of vulnerabilities that affect security of smart contracts, e.g., reentrancy bug [90], etc. Since many developers working on smart contract development are new in the area, they may not be aware of these vulnerabilities. There is also much code duplication in smart contracts [71]; copy-paste is a common development method. By copy-and-pasting, vulnerable code can easily "infect" other code. Thus, there is a need for tool supports to help developers not only to detect but also repair vulnerabilities to prevent them from "spreading" further.

Relatively mature bug finding and automated code inspection tools exist for conventional software, e.g., Findbugs[23], Facebook INFER [20], etc.; however these tools are not able to be used to statically check and identify smart contract vulnerabilities. Existing tools that detect smart contact bugs, e.g., Oyente [90], are relatively new, and much more study is needed to demonstrate their efficacy in terms of low false positive and false negative rate. Such studies have been done for Findbugs, and other tools for

[23]http://findbugs.sourceforge.net/

TABLE 7
Voting results of different demographic groups towards 28 challenges and desired improvements mentioned by interviewees. The **Total** row represents the number of respondents each group has. The rows **C1** to **I28** represent the percentages (%) of respondents from each demographic group who voted for 28 challenges and desired improvements; for example, the value 73.2 in the **C1** row means that 73.2% respondents from the **Dev** group (which has 190 respondents) rated C1 as one of top-3 major challenges they were facing during smart contract development.

| ID | Dev | Test | PM | seExpH | seExpM | seExpL | scExpH | scExpM | scExpL | Adv | nAdv | pubBlk | nPubBlk | bothBlk |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Total | 190 | 6 | 21 | 76 | 79 | 77 | 32 | 76 | 124 | 101 | 131 | 161 | 16 | 55 |
| C1 | 73.2 | 50.0 | 57.1 | 73.7 | 72.2 | 68.8 | 75.0 | 71.1 | 71.0 | 67.3 | 74.8 | 71.4 | 75.0 | 70.9 |
| C2 | 54.7 | 66.7 | 52.4 | 59.2 | 49.4 | 55.8 | 34.4 | 57.9 | 58.1 | 60.4 | 50.4 | 52.8 | 75.0 | 54.5 |
| C3 | 38.4 | 83.3 | 47.6 | 38.2 | 35.4 | 45.5 | 43.8 | 40.8 | 37.9 | 43.6 | 36.6 | 37.3 | 50.0 | 43.6 |
| C4 | 34.7 | 33.3 | 42.9 | 35.5 | 32.9 | 37.7 | 46.9 | 34.2 | 33.1 | 35.6 | 35.1 | 36.0 | 25.0 | 36.4 |
| C5 | 32.6 | 33.3 | 47.6 | 27.6 | 34.2 | 40.3 | 37.5 | 39.5 | 29.8 | 34.7 | 33.6 | 29.8 | 56.3 | 40.0 |
| C6 | 21.1 | 16.7 | 42.9 | 27.6 | 13.9 | 27.3 | 9.4 | 13.2 | 32.3 | 21.8 | 23.7 | 22.4 | 25.0 | 23.6 |
| C7 | 89.5 | 100.0 | 85.7 | 94.7 | 86.1 | 85.7 | 81.3 | 89.5 | 90.3 | 89.1 | 88.5 | 90.7 | 87.5 | 83.6 |
| C8 | 62.6 | 50.0 | 61.9 | 67.1 | 57.0 | 66.2 | 46.9 | 67.1 | 65.3 | 67.3 | 60.3 | 60.2 | 81.3 | 67.3 |
| C9 | 82.1 | 66.7 | 71.4 | 85.5 | 75.9 | 79.2 | 75.0 | 80.3 | 81.5 | 84.2 | 77.1 | 80.7 | 81.3 | 78.2 |
| C10 | 65.3 | 50.0 | 66.7 | 63.2 | 68.4 | 67.5 | 78.1 | 67.1 | 62.9 | 67.3 | 65.6 | 70.2 | 62.5 | 56.4 |
| C11 | 70.0 | 83.3 | 61.9 | 77.6 | 63.3 | 67.5 | 75.0 | 71.1 | 66.9 | 65.3 | 72.5 | 69.6 | 56.3 | 72.7 |
| C12 | 53.7 | 66.7 | 61.9 | 43.4 | 60.8 | 55.8 | 53.1 | 55.3 | 52.4 | 48.5 | 57.3 | 50.3 | 56.3 | 61.8 |
| C13 | 40.5 | 50.0 | 33.3 | 32.9 | 39.2 | 49.4 | 34.4 | 47.4 | 37.9 | 40.6 | 40.5 | 36.0 | 56.3 | 49.1 |
| C14 | 31.1 | 0.0 | 33.3 | 32.9 | 24.1 | 36.4 | 28.1 | 30.3 | 32.3 | 32.7 | 29.8 | 26.1 | 37.5 | 43.6 |
| C15 | 29.5 | 33.3 | 19.0 | 27.6 | 31.6 | 24.7 | 37.5 | 28.9 | 25.0 | 21.8 | 32.8 | 28.0 | 43.8 | 23.6 |
| C16 | 24.7 | 16.7 | 19.0 | 21.1 | 25.3 | 20.8 | 18.8 | 25.0 | 21.8 | 22.8 | 22.1 | 21.1 | 25.0 | 25.5 |
| C17 | 23.2 | 16.7 | 28.6 | 28.9 | 20.3 | 18.2 | 18.8 | 21.1 | 24.2 | 22.8 | 22.1 | 22.4 | 12.5 | 25.5 |
| I18 | 51.1 | 50.0 | 85.7 | 52.6 | 50.6 | 55.8 | 59.4 | 51.3 | 52.4 | 52.5 | 53.4 | 50.9 | 56.3 | 58.2 |
| I19 | 50.0 | 33.3 | 23.8 | 60.5 | 45.6 | 40.3 | 43.8 | 60.5 | 42.7 | 51.5 | 46.6 | 46.6 | 50.0 | 54.5 |
| I20 | 47.4 | 16.7 | 57.1 | 44.7 | 49.4 | 42.9 | 53.1 | 42.1 | 46.0 | 43.6 | 47.3 | 47.2 | 31.3 | 45.5 |
| I21 | 43.2 | 83.3 | 47.6 | 56.6 | 38.0 | 40.3 | 46.9 | 46.1 | 43.5 | 47.5 | 42.7 | 44.7 | 56.3 | 41.8 |
| I22 | 35.8 | 33.3 | 42.9 | 34.2 | 32.9 | 40.3 | 21.9 | 30.3 | 42.7 | 37.6 | 34.4 | 31.7 | 43.8 | 45.5 |
| I23 | 18.9 | 16.7 | 14.3 | 9.2 | 21.5 | 23.4 | 15.6 | 13.2 | 21.8 | 14.9 | 20.6 | 18.6 | 6.3 | 20.0 |
| I24 | 12.1 | 33.3 | 9.5 | 5.3 | 10.1 | 24.7 | 6.3 | 18.4 | 12.1 | 17.8 | 9.9 | 9.9 | 37.5 | 16.4 |
| I25 | 66.8 | 66.7 | 61.9 | 73.7 | 58.2 | 64.9 | 65.6 | 63.2 | 66.9 | 69.3 | 62.6 | 60.9 | 68.8 | 78.2 |
| I26 | 31.6 | 66.7 | 28.6 | 27.6 | 26.6 | 41.6 | 34.4 | 31.6 | 31.5 | 31.7 | 32.1 | 30.4 | 56.3 | 29.1 |
| I27 | 28.9 | 0.0 | 19.0 | 28.9 | 26.6 | 27.3 | 31.3 | 34.2 | 22.6 | 27.7 | 27.5 | 24.2 | 31.3 | 36.4 |
| I28 | 26.8 | 33.3 | 23.8 | 18.4 | 27.8 | 33.8 | 21.9 | 22.4 | 30.6 | 19.8 | 32.1 | 26.7 | 18.8 | 29.1 |

conventional software [120] but there are no similar studies yet for smart contract tools. More testing, fuzzing, and concolic testing tools can also be designed to augment existing static analysis tools to improve their efficacy. Besides, formal verification is also of great demand to reduce the possible adverse impact of smart contract vulnerabilities. Some researchers have tried to use formal verification methods to prove the correctness of smart contracts [61], [14], [4]. However, these approaches are still not mature yet and have not been demonstrated to scale to a large number of smart contracts of varying sizes.

Program repairs have recently become more mature with industrial adoption. Facebook is now using automated program repair to fix its apps[24]. Smart-contract specific repair solutions can also be designed to automatically patch vulnerabilities in smart contract. Generic repair is very difficult; but what works very well in practice are patching specific kinds of vulnerabilities. A novel program transformation tool extending existing tools that work for C and Java, such as Coccinelle [111] or Spoon [115], can potentially be designed for Solidity. Next, required transformations can be specified as semantic patches [104] and applied to patch existing smart contracts that suffer from vulnerabilities.

Developers also mentioned bugs in Solidity compiler. This is a serious issue since such bugs can translate to vulnerabilities and unreliable executions of many smart contracts. More mature compilers like gcc have shown to be buggy [138] and compiler testing solutions have found many of such bugs. There is a need for a further study to demonstrate the extent existing compiler testing solutions can work to identify bugs in Solidity compilers and design steps to adapt the solutions for them to be more effective for Solidity.

Developers are also in need for best practices and code smells that may prevent them from introducing vulnerabilities. Systematic literature review and cataloging of such vulnerabilities is one first step. Designing common repositories to store common vulnerabilities specific to smart contact – in similar fashion like CVE[25] – is another step. Operationalizing CVE into tools, e.g., [132] is yet another step.

## 5.2 Other Factors Affecting Smart Contract Development

Aside from security, many other factors affect smart contract development. Here, we highlight five different aspects of smart contract development that pose open research problems requiring advances in the field.

### *5.2.1 Programming Language and Virtual Machine Design*

Solidity and Ethereum VM are in their infancy and developers often encounter difficulties in developing smart contracts due to their limitations (e.g., type checking, memory management, multi-threading support, etc.). These highlights opportunities for research to add additional features in Solidity and Ethereum VM. Additional consideration needs to be put in the design of these features considering specific constraints for smart contract and the unique way it is deployed and run in a distributed manner. For example, adding multi-threading support to Ethereum VM is non-trivial. “A miner cannot simply execute these contracts in

[24]https://code.fb.com/developer-tools/finding-and-fixing-software-bugs-automatically-with-sapfix-and-sapienz/

[25]http://cve.mitre.org/

parallel, because they may perform conflicting accesses to shared data, and an arbitrary interleaving could produce an inconsistent final state." [39] We have seen recent early work proposing these missing features, e.g., [33], [116], that require deep technical novelty as "porting" features from a popular programming language (e.g., Java) to Solidity and a popular VM (e.g., Java VM) to Ethereum VM are non-trivial. Existing proposed research solutions often have trade-off or introduce additional complexities that may prevent their adoptions; further research is needed to develop additional solutions that may consider other trade-offs to help Ethereum VM and Solidity language designers/maintainers decide the most promising approach or direction that the community should take. These decisions need to be taken carefully as it will have long term implications.

Another possible direction is to enable developers to code in their language of choice (or a restricted subset of that language) and allow their code to be translated to Solidity. Recent research have explored ways to transform Java to C# [107], [157]. There may also be solutions developed to transform code written in languages such as Javascript (which has a large developer base, and is similar to Solidity) to Solidity code.

#### 5.2.2 Better Resource Management

Smart contract developers need to optimize for gas and efficiency while constrained with stack size, number of local variables, etc. This makes it harder for developers to focus on designing cool new features. Manual optimization of these considerations also pose other issues (e.g., readability). Thus, new support to help developers optimize for gas considering the various constraints is needed. Current solution only provides estimate for bytecode but developers may need support for source code and developers may also need recommendations on ways to optimize code. Solutions that can automatically and safely transform a code that is readable (to developers) but do not satisfy constraints into another code that satisfy constraints (but is less readable) seamlessly may also be in demand. These solutions are non-trivial and further research is needed in these directions. Existing research on program transformations, e.g., [141], [91], [99] can be a good starting point in designing these solutions.

#### 5.2.3 Library Construction

Developers are in serious need of libraries. The level of code redundancies among deployed smart contracts is high – this highlights that developers are reinventing-the-wheel often. This is not surprising as modern software are often built on top of libraries – for example, libraries can comprise of more than 90% of a web application[26]. Tools are needed to identify reusable common components used in many smart contracts and organize them into easy-to-find and easy-to-use classes, methods, and libraries. Methods from clone detection [72], [148] and code categorization [102], [73], [130] can potentially be employed to construct such libraries. Security considerations need to also be considered in the construction of such libraries to ensure that vulnerabilities do not spread through library dependencies, c.f., [36].

#### 5.2.4 Evolution, Maintenance, and Deployment of Smart Contracts

As one developer mentions, once a smart contract is deployed it is not possible for it to be modified. There are workarounds to address the evolution of contracts (with varying levels of difficulty and impact to users), such as by using *detegatecall* (i.e., separating data and logic of a smart contract in separate contracts and letting the data contract call the logic contract through *delegatecall*)[27], or using a registry contract to store latest version of a contract[28], etc. However, no systematic study has been done on the advantages and disadvantages of different maintenance options. Further study is needed to explore this and to possibly develop new maintenance, evolution, and deployment methods that prevent smart contract evolution to adversely affect developers and users.

Solidity API also changes frequently and are often not backward compatible. There are needs to help developers evolve Solidity code to "catch up" with API evolution. Studies are needed to explore if existing research solutions [57], [63] work well for Solidity API and if not, novels solutions are needed. Developers also expressed their desire for up-to-date documentations for tools/languages they are using. It would be valuable to find proper approaches that can help automatically update or even generate documentation and link pieces of documentation from different sources on the internet. Ideas from exiting studies that recommending adaptive changes for documentation evolution [35] and detecting API documentation errors [156] may help in this direction.

#### 5.2.5 Supporting End-Users

Smart contract technology has a great potential in the financial domain while its development is still nascent. To facilitate the widespread adoption of smart contracts in finance industry, some big investment banks (e.g., European Bank for Reconstruction and Development (EBRD)), International Swaps and Derivatives Association (ISDA), leading law firms (e.g., Linklaters), as well as some researchers, have been working to develop a set of best practice and industry-wide standards in terms of the construction, execution, and validation of smart contracts from both legal and technical perspectives [65], [66], [32], [28], [29]. Related to the construction of smart contracts, a noticeable problem is that developers of smart contracts may not be finance domain experts. There is a need to enable financial experts to write smart contracts directly without intermediaries. Intermediaries may introduce miscommunication and bugs [1]. Simplified domain-specific languages have been designed for many areas, e.g., ABB have designed a simplified language to enable end users to directly program robots with more ease [131]. Similar solutions can also be designed for smart contract development. These solutions may involve design of a specialized Domain Specific Language, text-to-code solutions, program synthesis from examples, bots that can clarify requirements from financial experts, etc. Prior work have shown that these technologies are feasible for specialized domains [150], [8], but more effort needs to be invested in their design.

## 6 Threats to Validity

**Internal Validity** In our paper, we designed our survey questions based on the interview results. However, it is possible that we may draw wrong conclusions from interviewees' comments. To alleviate this threat, we tried to read the interview transcription several times; and each step of card sorting of interview comments

[26]http://www.linkedin.com/pulse/how-can-you-ensure-your-open-source-components-secure-sharma/

[27]http://blog.trailofbits.com/2018/09/05/contract-upgrade-anti-patterns/

[28]https://ethereum.stackexchange.com/questions/2404/upgradeable-smart-contracts?noredirect=1\&lq=1

was performed and verified by two researchers. Besides, before sending the questions to our potential survey respondents, we conducted a pilot study in which we asked 5 developers to fill the survey and collected their feedbacks on the questions and answer options. Refinement is subsequently made based on developers' comments.

It is also possible that survey respondents may have provided dishonest answers (e.g., saying what they want us to hear or saying what we want to hear) due to various reasons. To help reduce this bias, we made the following efforts: (1) In our survey invitation letter, we explicitly mentioned that no personal information would be disseminated in our paper. (2) We allowed our survey respondents to be anonymous; they are untraceable if they do not leave email addresses; and they can also leave new/anonymous email addresses. According to [110], confidentiality and anonymity helped in obtaining un-biased answers from survey respondents.

Besides, following the advice in [78], i.e. using the proper language medium for intended respondents, we also translated our survey into Chinese to ensure that respondents from China can understand our survey questions well. We only have our survey in English and Chinese since English is a lingua franca and Chinese is the most spoken language in the world. During result analysis, similarly, it is also possible that we may draw wrong conclusions about survey respondents' perceptions based on their comments. To alleviate this threat, we also tried to read the survey comments several times.

**External Validity** Following the strategy of previous studies [134], [6], we stopped our interviews when we reached the saturation of findings after interviewing 20 persons (this number was also similar as prior studies [94], [60]). We have to admit that the notion of "saturation of findings" may introduce interviewer subjectivity and risk missing information. To avoid these problems, we tried to include two interviewers for each interview based on both interviewees' and interviewers' schedules. In total, 15 out of the 20 interviews were conducted by two interviewers. They worked together to take memos and asked questions during interviews. Having two interviewers could help us: (1) capture as much relevant information as possible during interviews (some information may be missed by single interviewer), and (2) reduce the chance of unfair subjective bias in the discussion of whether the saturation of findings has been reached (we had a more comprehensive note for comparison with previous interview notes, and by having multiple interviewers, collectively, we could better recall the details of what happened or what was discussed).

Considering there may exist other populations who might add new insights, we also need to acknowledge that the opinions provided by our interviewees may not be representative of and agreeable to the whole community. To reduce this threat, we ensured that our interviewees hold various roles and have different levels of expertise, e.g., developers, trainers, CEOs in companies developing smart contract applications, etc. We believe that their comments still uncovered various insights into the challenges of smart contract development.

To validate our interview findings, we conducted a survey with 232 developers from 48 countries. As our respondents were mainly recruited through GitHub, we may risk ignoring some developers (e.g., from proprietary smart contract development) who are unlikely or are not permitted to respond to our survey. Thus we cannot guarantee that our findings could be generalized to all relevant smart contract practitioners. However, our respondents had different experience levels, educational qualifications, and contribute to various projects (including open-source and close-source projects) on different blockchain platforms (including public and non-public blockchains). Such a diversity in backgrounds to a large extent, made us believe that our survey results still projected valuable insights into the challenges of smart contract development. To further improve the generalizability of our findings, we encourage other researchers to replicate our study with more developers in the future.

## 7 Related Work

In this section, we highlight related work on smart contract, including empirical studies on smart contract, tools developed for smart contract, and studies on challenges and opportunities in other domains outside of smart contract.

**Empirical Studies on Smart Contract:** The rapid growth of smart contract development motivated a series of empirical studies. These studies mainly aimed to explore the characteristics and potential impact of smart contracts [11], [69], summarize development patterns or lessons [151], [37], evaluate existing programming languages and techniques [112], [53], propose some feasible strategies for smart contract programming and altering/undoing [1], [64], [96], etc.

Bartoletti et al. studied the application domains and design of 834 verified smart contracts from Bitcoin and Ethereum [11]. Fröwis et al. investigated the problem of control flow immutability of smart contacts on Ethereum [45]. Bartoletti et al. analyzed how smart contract can be used to implement Ponzi schemes on Ethereum [9] while Juels et al. mentioned the feasibility of performing criminal activities on Ethereum, e.g., leaking secret documents [69].

Delmolino et al. summarized some common mistakes that students made during smart contract programming classes and provided a guide to help people avoid those mistakes [37]. By applying grounded theory into collected smart contract data, Wohrer et al. summarized some security patterns and corresponding solutions [151]. Unterweger et al. presented some lessons they learned during their implementation of a privacy-preserving smart contract in the energy domain [145].

Parizi et al. did an evaluation of usability and security of smart contract programming languages [112]. They also did an assessment over existing smart contract testing techniques on Ethereum [113]. Grishchenko et al. did an overview of various static analysis tools that can be applied to smart contracts, covering formal semantics, security definitions, and verification tools [53]. Miller et al. provided an overview of existing smart contract languages and tools for analyzing smart contracts; they also presented some research challenges for formal verification methods and program analysis applied to smart contracts [100].

Idelberger et al. studied the utility of logic-based smart contracts and explored how they could be used in blockchains [64]. Sergey and Hobor suggested to use existing formal methods to reason about concurrency of smart contract [128]. Marino and Juels developed a set of standards for altering and undoing smart contacts [96]. Khalil et al. suggested that more attention being paid to the traditional developers (i.e., the lawyers) of contracts [1]. Clack et al. argued that a formal language which handles over-the-counter financial smart contract derivatives needs to combine temporal, deontic and operational aspects for such a formalism [31]. Destefanis et al. called for a definition of blockchain software engineering to help solve/avoid some smart contract issues [38].

Unlike the above studies that mainly focus on performing empirical analyses on specific aspects of smart contracts (e.g., specific application domains or security patterns), our study explored the major challenges developers are facing during smart contract development. Through interviews and a follow-up survey, we identified several major barriers that prevent developers from effectively developing smart contracts.

**Tools for Smart Contract:** Various kinds of tools have been proposed to resolve smart contract related problems, ranging from detecting bugs [90], [50], [4], [61], guarding data privacy/quality [154], [123], to easing smart contract creation [44], [97] and manual analysis [158], [15].

Due to the nature of smart contracts, bugs tend to be costly, thus substantial efforts have been made to detect vulnerabilities of smart contracts or to prove the correctness of smart contracts. Luu et al. developed Oyente to identify several pre-defined kinds of security bugs (such as transaction order dependency) [90]. Nikolic et al. developed MAIAN to identify greedy, prodigal, and suicidal smart contracts [108]. Liu et al. developed ReGuard to detect reentrancy bugs through fuzz testing [87]. Chen et al. and Grech et al. attempted to identify gas-related problems [22], [23], [50]. Marescotti et al. further proposed two approaches inspired by model-checking techniques to compute the exact worst-case gas consumption for smart contracts [95]. Chen et al. proposed a method to detect potential Ponzi schemes on Ethereum [24]. Tsankov et al. developed Securify to detect several kinds of security bugs by inspecting whether or not smart contract behavior violated certain semantic patterns derived from control- and data-flow dependencies within smart contract [144], [143]. Grishchenko et al. and Grossman et al. tried to detect vulnerabilities through reachability analysis [51] and effective-callback-free objects detection [54]. Jiang et al. developed ContractFuzzer to detect security vulnerabilities through generating fuzzing inputs and instrumenting EVM [67]. Liu et al. attempted to predict potential vulnerabilities by identifying irregular token sequences [88]. Tikhomirov et al. developed SmartCheck to detect potential problems by checking against XPath patterns [142]. Krupp et al. developed TEETHER to automatically generate an exploit for a smart contract given its binary code [81]. Wang et al. proposed a random based and a NSGA-II based multi-objective approach to generate cost-effective test suites for smart contracts [149]. They further explored the potential of applying mutation testing into smart contracts [153], [84].

To prove the correctness of smart contracts, some researchers proposed to use formal verification methods to perform complete analysis of smart contracts by using interactive theorem provers [13], [61], [14], [109], [4], such as Isabelle/HOL[29], F*[30], Why3 [42], and K[31], etc. Recently, Grishchenko et al. has formalized a complete small-step semantics of EVM bytecode for the F* proof assistant [52]. Rosu et al. also developed KEVM, a formal semantics of the EVM in the K framework [121], [25]; and further evaluated its effectiveness in verifying EVM smart contracts [59], [114]. Sergey et al. proposed a verification framework based on Scilla (an intermediate representation languages specifically designed for verification) to apply formal verification methods to reason about temporal properties of smart contracts [129]. Alt. et al. built an SMT-based formal verification module inside the Solidity compiler, where during compilation, users could get automatic warnings of and counterexamples for several kinds of potential problems like unreachable code, assertion failures, etc. [3]. Hirai used Kripke models of the modal logic to check the atomicity property of a protocol called "atomic cross-chain swap" (expressed in a form of hashed timelock smart contracts) [62]. Besides formal verification, some researchers proposed to abstract smart contacts to a certain form before conducting relevant verification tasks [71], [50].

To guard the quality or confidentiality of data involved in smart contract execution, Zhang et al. designed a data feed system called Town Crier [154] to provide trusted input data for smart contracts and keeping data requests secret from others. Sánchez and Cerezo proposed a system called Raziel to help securely executing smart contracts while guaranteeing their privacy, correctness, and verifiability [123]. Liang et al. proposed a framework called DESC to automatically control access in the domain of secure data exchange and protect data owners' rights [85].

To ease smart contract creation, Frantz and Nowostawski proposed to semi-automatically create smart contracts by translating textual contract into smart contract rules [44]. Mavridou et al. proposed a framework called FSolidityM to allow developers to design smart contract as Finite State Machines [97]. Schrans et al. invented a programming language called Flint introducing caller capabilities, and safe atomic operations [125]. Seijas et al. explored the design of Marlowe, a domain specific language targeted at financial contracts on blockchains, together with examples of its use; they further described a tool, called Meadow, that allows users to interact with and simulate the operations of Marlowe contracts [126]. Valliappan et al. combined Simplicity (a language for programming smart contracts with a formal semantic) with a categorical model, to facilitate the addition of local definitions, functions, and bounded loops [147]. Bartoletti et al. designed a high-level domain specific language with a computationally sound compiler, namely BitML, for Bitcoin smart contracts. BitML creates smart contracts in the form of symbolic expressions, then compiles these expressions to Bitcoin scripts [10], [12]. To help people better understand and analyze smart contracts, Brent et al. proposed a framework called Vandal that decompiled EVM bytecode and allowed developers to analyze bytecode via logic specification [16]. Zhou et al. developed Erays to generate high-level pseudocode from binary code of smart contracts [158]. Bragagnolo et al. developed SmartInspect which allowed users to understand contract stored state without redeploying a smart contract [15]. Additionally, Dickerson et al. proposed a way to allow smart contracts to be executed in parallel by adapting techniques from software transactional memory [39]. Colombo et al. developed CONTRACTLARVA to recover smart contract from violations dynamically [34].

Unlike the above studies which aimed to develop specific tools/techniques for smart contract, we focused on identifying major challenges developers are facing during smart contract development. Our study also identified several kinds of tools that developers desired most, such as advanced debuggers, source-code-level gas estimations, advanced formal verification techniques, etc. Our study provides a guide for tool builders to develop tools that are needed by developers.

**Studies on Challenges and Opportunities:** There have been several papers studying the challenges and corresponding opportunities in specific domains or software practices. Two studies done by Porru et al. [117] and Lin et al. [86] are mostly re-

[29]Isabelle. http://isabelle.in.tum.de/
[30]F*. https://www.fstar-lang.org/
[31]K Framework. http://www.kframework.org/index.php/Main_Page

lated to our study. Porru et al. [117] studied the challenges and new directions of blockchain-oriented software engineering, from defining new professional roles, enhancing security and reliability, to developing novel tools for software architecture/modeling, ensuring effective testing activities, etc. Their study mainly discussed some high-level challenges/directions in developing blockchain-oriented software, including both blockchain platforms and general blockchain applications. Unlike their study, our work specifically studied the challenges of smart contract (a special kind of blockchain application) development from the practitioners' view. Our study provided some concrete and actionable directions for both researchers and practitioners to take on to facilitate the development of smart contracts. Lin et al. [86] briefly summarized some issues and challenges that people need to concern when trying to embrace the blockchain technologies, e.g., regulations problems, scale of blockchain problems, etc. Unlike them, we did not study the adoption of blockchain itself; instead, we focused on exploring the challenges and opportunities of developing smart contracts which run on blockchain platforms.

Zhang et al. did a survey of cloud computing technology and presented some design challenges of cloud computing [155]. Similarly, Dillon et al. presented several challenges from the cloud computing adoption perspective and figured out that the cloud interoperability issue deserved substantial attention [40]. Kephart [74] outlined some scientific and engineering challenges of autonomic computing. Labrinidis and Jagadish discussed some controversies and myths surrounding big data and summarized some challenges and opportunities with big data [82]. Manferdelli listed some challenges and opportunities during software development based on many-core computing [93].

Knight summarized some challenges and directions in developing safety critical systems [79]. Broy studied the challenges in automotive software engineering [18]. Muccini et al. [103] and Joorabchi et al. [68] explored relevant challenges in software testing and software development of mobile applications respectively. Hilton et al. investigated the barriers and unmet needs faced by developers during their adoption of continuous integration systems [60]. Gousios et al. studied work practices and challenges in pull-based development from both the contributor's and integrator's perspective [48], [49]. Kim et al. did an empirical study to understand refactoring challenges and benefits at Microsoft [77].

Unlike these studies, we explored challenges and opportunities of a *new* topic, i.e., the development of smart contracts. We summarized six major categories of challenges and further identified some potential research directions specific to the smart contract domain.

## 8 Conclusion and Future Work

Smart contract, which originally refers to the automation of legal contracts in general, has recently seen much interest due to the rise of blockchain technology. Today, it is popularly used to refer to low-level code scripts running on blockchains. In this study, we investigated the challenges developers are facing in developing such smart contracts, especially focusing on the Ethereum platform. Our interview and survey results indicate that smart contract development is still in its infancy: there is no generally accepted way to secure smart contract code; the existing development toolchain is not powerful enough; development and runtime platforms (i.e., programming languages, virtual machines) still have a lot of limitations; online learning resources and community supports are limited. Based on our findings, we summarized some concrete and actionable directions in which researchers and practitioners could take on in the future (e.g., automated smart contract patching, Solidity compiler testing, source-code-level gas optimization, automated Solidity library construction, etc.). Progress in such directions would further facilitate smart contract development.

## References


[1] F. Al Khalil, T. Butler, L. O'Brien, and M. Ceci. Trust in smart contracts is a process, as well. In *Proceedings of the 21st International Conference on Financial Cryptography and Data Security*, pages 510–519, 2017.

[2] M. Alharby and A. van Moorsel. Blockchain-based Smart Contracts: A Systematic Mapping Study. *arXiv preprint arXiv:1710.06372*, 2017.

[3] L. Alt and C. Reitwiessner. SMT-based verification of Solidity smart contracts. In *Proceedings of International Symposium on Leveraging Applications of Formal Methods*, pages 376–388, 2018.

[4] S. Amani, M. Bégel, M. Bortin, and M. Staples. Towards verifying Ethereum smart contract bytecode in Isabelle/HOL. In *Proceedings of the 7th International Conference on Certified Programs and Proofs*, pages 66–77, 2018.

[5] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, volume 13, 2013.

[6] M. Aniche, C. Treude, I. Steinmacher, I. Wiese, G. Pinto, M.-A. Storey, and M. A. Gerosa. How modern news aggregators help development communities shape and share knowledge. In *Proceedings of the 40th International Conference on Software Engineering*, pages 499–510, 2018.

[7] ARM. Arm security technology – building a secure system using trustzone technology. *ARM Technical White Paper*, http://infocenter.arm.com/help/topic/com.arm.doc.prd29genc-009492c/PRD29GENC009492C_trustzone_security_whitepaper.pdf, 2009.

[8] D. W. Barowy, S. Gulwani, T. Hart, and B. Zorn. FlashRelate: Extracting relational data from semi-structured spreadsheets using examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 218–228, 2015.

[9] M. Bartoletti, S. Carta, T. Cimoli, and R. Saia. Dissecting Ponzi schemes on Ethereum: Identification, analysis, and impact. *arXiv preprint arXiv:1703.03779*, 2017.

[10] M. Bartoletti, T. Cimoli, and R. Zunino. Fun with Bitcoin smart contracts. In *Proceedings of International Symposium on Leveraging Applications of Formal Methods*, pages 432–449, 2018.

[11] M. Bartoletti and L. Pompianu. An empirical analysis of smart contracts: Platforms, applications, and design patterns. In *Proceedings of the 21st International Conference on Financial Cryptography and Data Security*, pages 494–509, 2017.

[12] M. Bartoletti and R. Zunino. BitML: A calculus for Bitcoin smart contracts. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security*, pages 83–100, 2018.

[13] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, et al. Formal verification of smart contracts: Short paper. In *Proceedings of the Workshop on Programming Languages and Analysis for Security@CCS 2016*, pages 91–96, 2016.

[14] G. Bigi, A. Bracciali, G. Meacci, and E. Tuosto. Validation of decentralised smart contracts through game theory and formal methods. In *Programming Languages with Applications to Biology and Security*, pages 142–161. Springer, 2015.

[15] S. Bragagnolo, H. Rocha, M. Denker, and S. Ducasse. *Smartinspect: Smart contract inspection technical report*. PhD thesis, Inria Lille, 2017.

[16] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981*, 2018.

[17] R. G. Brown, J. Carlyle, I. Grigg, and M. Hearn. Corda: An introduction. *R3 CEV, August*, 2016.

[18] M. Broy. Challenges in automotive software engineering. In *Proceedings of the 28th International Conference on Software Engineering*, pages 33–42, 2006.

[19] V. Buterin. A next-generation smart contract and decentralized application platform. *white paper*, 2014.

[20] C. Calcagno and D. Distefano. Infer: An automatic program verifier for memory safety of C programs. In *Proceedings of the 3rd International Symposium on NASA Formal Methods*, pages 459–465, 2011.

[21] T. Chen, X. Li, X. Luo, and X. Zhang. Under-optimized smart contracts devour your money. In *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering*, pages 442–446, 2017.

[22] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, M. H. Au, and X. Zhang. An adaptive gas cost mechanism for Ethereum to defend against under-priced DoS attacks. In *Proceedings of the 13th International Conference on Information Security Practice and Experience*, pages 3–24, 2017.

[23] T. Chen, Z. Li, H. Zhou, J. Chen, X. Luo, X. Li, and X. Zhang. Towards saving money in using smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 81–84, 2018.

[24] W. Chen, Z. Zheng, J. Cui, E. Ngai, P. Zheng, and Y. Zhou. Detecting Ponzi schemes on Ethereum: Towards healthier blockchain technology. In *Proceedings of the 27th World Wide Web Conference on World Wide Web*, pages 1409–1418, 2018.

[25] X. Chen, D. Park, and G. Roşu. A language-independent approach to smart contract verification. In *Proceedings of International Symposium on Leveraging Applications of Formal Methods*, pages 405–413, 2018.

[26] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. *arXiv preprint arXiv:1804.05141*, 2018.

[27] C. D. Clack. Smart contract templates: Legal semantics and code validation. *Journal of Digital Banking*, 2(4):338–352, 2018.

[28] C. D. Clack. Smart contract templates: The semantics of smart legal agreements. *Journal of Digital Banking*, 2(4):1–15, 2018.

[29] C. D. Clack, V. A. Bakshi, and L. Braine. Smart contract templates: Foundations, design landscape and research directions. *arXiv preprint arXiv:1608.00771*, 2016.

[30] C. D. Clack and C. McGonagle. Smart derivatives contracts: The ISDA Master Agreement and the automation of payments and deliveries. *arXiv preprint arXiv:1904.01461*, 2019.

[31] C. D. Clack and G. Vanca. Temporal aspects of smart contracts for financial derivatives. In *Proceedings of International Symposium on Leveraging Applications of Formal Methods*, pages 339–355, 2018.

[32] Clifford Chance and European Bank for Reconstruction and Development. Smart contracts: Legal framework and proposed guidelines for lawmakers. 2017.

[33] M. Coblenz. Obsidian: A safer blockchain programming language. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 97–99, 2017.

[34] C. Colombo, J. Ellul, and G. J. Pace. Contracts over smart contracts: Recovering from violations dynamically. In *Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 300–315, 2018.

[35] B. Dagenais and M. P. Robillard. Using traceability links to recommend adaptive changes for documentation evolution. *IEEE Transactions on Software Engineering*, 40(11):1126–1146, 2014.

[36] A. Decan, T. Mens, and E. Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 181–191, 2018.

[37] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *Proceedings of the 20th International Conference on Financial Cryptography and Data Security*, pages 79–94, 2016.

[38] G. Destefanis, M. Marchesi, M. Ortu, R. Tonelli, A. Bracciali, and R. Hierons. Smart contracts vulnerabilities: A call for blockchain software engineering? In *International Workshop on Blockchain Oriented Software Engineering@SANER*, pages 19–25, 2018.

[39] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen. Adding concurrency to smart contracts. In *Proceedings of the 36th Symposium on Principles of Distributed Computing*, pages 303–312, 2017.

[40] T. Dillon, C. Wu, and E. Chang. Cloud computing: Issues and challenges. In *Proceedings of the 24th International Conference on Advanced Information Networking and Applications*, pages 27–33, 2010.

[41] J. Eberhardt and S. Tai. On or off the blockchain? Insights on off-chaining computation and data. In *Proceedings of the 6th European Conference on Service-Oriented and Cloud Computing*, pages 3–15. Springer, 2017.

[42] J.-C. Filliâtre and A. Paskevich. Why3--Where programs meet provers. In *Proceedings of the 22nd European Symposium on Programming*, pages 125–128, 2013.

[43] R. A. Fisher. On the interpretation of $\chi$ 2 from contingency tables, and the calculation of P. *Journal of the Royal Statistical Society*, 85(1):87–94, 1922.

[44] C. K. Frantz and M. Nowostawski. From institutions to code: Towards automated generation of smart contracts. In *Proceedings of the 1st International Workshops on Foundations and Applications of Self* Systems*, pages 210–215, 2016.

[45] M. Fröwis and R. Böhme. In code we trust? In *ESORICS 2017 International Workshops on Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 357–372. Springer, 2017.

[46] H. S. Galal and A. M. Youssef. Trustee: Full privacy preserving vickrey auction on top of ethereum. *arXiv preprint arXiv:1905.06280*, 2019.

[47] L. Goodman. Snowball sampling. *Annals of Mathematical Statistics*, 32(1):148–170, 1961.

[48] G. Gousios, M.-A. Storey, and A. Bacchelli. Work practices and challenges in pull-based development: The contributor's perspective. In *Proceedings of the 38th International Conference on Software Engineering*, pages 285–296, 2016.

[49] G. Gousios, A. Zaidman, M.-A. Storey, and A. Van Deursen. Work practices and challenges in pull-based development: The integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering*, pages 358–368, 2015.

[50] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis. Madmax: Surviving out-of-gas conditions in Ethereum smart contracts. In *Proceedings of the ACM on Programming Languages, 2(OOPSLA)*, pages 14–18, 2018.

[51] I. Grishchenko, M. Maffei, and C. Schneidewind. EtherTrust: Sound static analysis of Ethereum bytecode.

[52] I. Grishchenko, M. Maffei, and C. Schneidewind. A semantic framework for the security analysis of Ethereum smart contracts. In *Proceedings of the 7th International Conference on Principles of Security and Trust*, pages 243–269, 2018.

[53] I. Grishchenko, M. Maffei, and C. Schneidewind. Foundations and tools for the static analysis of Ethereum smart contracts. In *Proceedings of the 30th International Conference on Computer Aided Verification*, pages 51–78, 2018.

[54] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar. Online detection of effectively callback free objects with applications to smart contracts. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–28, 2017.

[55] G. Guest, A. Bunce, and L. Johnson. How many interviews are enough? An experiment with data saturation and variability. *Field methods*, 18(1):59–82, 2006.

[56] P. Hegedus. Towards analyzing the complexity landscape of Solidity based Ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 35–39, 2018.

[57] J. Henkel and A. Diwan. CatchUp! Capturing and replaying refactorings to support API evolution. In *Proceedings of the 27th International Conference on Software Engineering*, pages 274–283, 2005.

[58] J. Henrich, S. J. Heine, and A. Norenzayan. The weirdest people in the world? *Behavioral and brain sciences*, 33(2-3):61–83, 2010.

[59] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, et al. KEVM: A complete formal semantics of the Ethereum Virtual Machine. In *Proceedings of the 31st Computer Security Foundations Symposium*, pages 204–217, 2018.

[60] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig. Trade-offs in continuous integration: Assurance, security, and flexibility. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, pages 197–207, 2017.

[61] Y. Hirai. Formal verification of Deed contract in Ethereum name service, 2016.

[62] Y. Hirai. Blockchains as Kripke models: An analysis of atomic cross-chain swap. In *Proceedings of International Symposium on Leveraging Applications of Formal Methods*, pages 389–404, 2018.

[63] A. Hora and M. T. Valente. apiwave: Keeping track of API popularity and migration. In *Proceedings of the 31st International Conference on Software Maintenance and Evolution*, pages 321–323, 2015.

[64] F. Idelberger, G. Governatori, R. Riveret, and G. Sartor. Evaluation of logic-based smart contracts for blockchain systems. In *Proceedings of the 10th International Symposium on Rules and Rule Markup Languages for the Semantic Web*, pages 167–183, 2016.

[65] ISDA and King & Wood Mallesons. Smart derivatives contracts: From concept to construction. 2018.

[66] ISDA and Linklaters. Smart contracts and distributed ledger - a legal perspective. 2017.

[67] B. Jiang, Y. Liu, and W. Chan. ContractFuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd International Conference on Automated Software Engineering*, pages 259–269, 2018.

[68] M. E. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile app development. In *Proceedings of the 7th International Symposium on Empirical Software Engineering and Measurement*, pages 15–24, 2013.

[69] A. Juels, A. Kosba, and E. Shi. The ring of gyges: Investigating the future of criminal smart contracts. In *Proceedings of the 23rd Conference on Computer and Communications Security*, pages 283–295, 2016.

[70] H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten. Arbitrum: Scalable, private smart contracts. In *Proceedings of the 27th {USENIX} Security Symposium*, pages 1353–1370, 2018.

[71] S. Kalra, S. Goel, M. Dhawan, and S. Sharma. Zeus: Analyzing safety of smart contracts. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, 2018.

[72] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[73] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. Mudablue: An automatic categorization system for open source repositories. *Journal of Systems and Software*, 79(7):939–953, 2006.

[74] J. O. Kephart. Research challenges of autonomic computing. In *Proceedings of the 27th international conference on Software engineering*, pages 15–22, 2005.

[75] M. Kim, T. Zimmermann, R. DeLine, and A. Begel. The emerging role of data scientists on software development teams. In *Proceedings of the 38th International Conference on Software Engineering*, pages 96–107, 2016.

[76] M. Kim, T. Zimmermann, R. DeLine, and A. Begel. Data scientists in software teams: State of the art and challenges. *IEEE Transactions on Software Engineering*, (1):1–1, 2017.

[77] M. Kim, T. Zimmermann, and N. Nagappan. An empirical study of refactoringchallenges and benefits at Microsoft. *IEEE Transactions on Software Engineering*, 40(7):633–649, 2014.

[78] B. A. Kitchenham and S. L. Pfleeger. Personal opinion surveys. In *Guide to advanced empirical software engineering*, pages 63–92. 2008.

[79] J. C. Knight. Safety critical systems: Challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, pages 547–550, 2002.

[80] P. S. Kochhar, X. Xia, D. Lo, and S. Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 165–176, 2016.

[81] J. Krupp and C. Rossow. TEETHER: Gnawing at Ethereum to automatically exploit smart contracts. In *Proceedings of the 27th USENIX Security Symposium*, pages 1317–1333, 2018.

[82] A. Labrinidis and H. V. Jagadish. Challenges and opportunities with big data. *The VLDB Endowment*, 5(12):2032–2033, 2012.

[83] C. Li, B. Palanisamy, and R. Xu. Scalable and privacy-preserving design of on/off-chain smart contracts. *arXiv preprint arXiv:1902.06359*, 2019.

[84] Z. Li, H. Wu, J. Xu, X. Wang, L. Zhang, and Z. Chen. MuSC: A Tool for mutation testing of Ethereum smart contract. In *Proceedings of the 34th International Conference on Automated Software Engineering-Demonstrations (Accepted)*, 2019.

[85] J. Liang, W. Han, Z. Guo, Y. Chen, C. Cao, X. S. Wang, and F. Li. DESC: Enabling secure data exchange based on smart contracts. *Science China Information Sciences*, 61(4):049102, 2018.

[86] I.-C. Lin and T.-C. Liao. A survey of blockchain security issues and challenges. *IJ Network Security*, 19(5):653–659, 2017.

[87] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe. ReGuard: Finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering Companion*, pages 65–68, 2018.

[88] H. Liu, C. Liu, W. Zhao, Y. Jiang, and J. Sun. S-gram: Towards semantic-aware security auditing for Ethereum smart contracts. In *Proceedings of the 33rd International Conference on Automated Software Engineering*, pages 814–819, 2018.

[89] D. Lo, N. Nagappan, and T. Zimmermann. How practitioners perceive the relevance of software engineering research. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, pages 415–425. ACM, 2015.

[90] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 23rd Conference on Computer and Communications Security*, pages 254–269, 2016.

[91] S. Ma, D. Lo, T. Li, and R. H. Deng. Cdrep: Automatic repair of cryptographic misuses in android applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 711–722, 2016.

[92] D. Magazzeni, P. McBurney, and W. Nash. Validation and verification of smart contracts: A research agenda. *Computer*, 50(9):50–57, 2017.

[93] J. L. Manferdelli, N. K. Govindaraju, and C. Crall. Challenges and opportunities in many-core computing. *Proceedings of the IEEE*, 96(5):808–815, 2008.

[94] I. Manotas, C. Bird, R. Zhang, D. Shepherd, C. Jaspan, C. Sadowski, L. Pollock, and J. Clause. An empirical study of practitioners' perspectives on green software engineering. In *Proceedings of the 38th International Conference on Software Engineering*, pages 237–248, 2016.

[95] M. Marescotti, M. Blicha, A. E. Hyvärinen, S. Asadi, and N. Sharygina. Computing exact worst-case gas consumption for smart contracts. In *Proceedings of International Symposium on Leveraging Applications of Formal Methods*, pages 450–465, 2018.

[96] B. Marino and A. Juels. Setting standards for altering and undoing smart contracts. In *Proceedings of the 10th International Symposium on Rules and Rule Markup Languages for the Semantic Web*, pages 151–166, 2016.

[97] A. Mavridou and A. Laszka. Designing secure Ethereum smart contracts: A finite state machine based approach. *arXiv preprint arXiv:1711.09327*, 2017.

[98] J. H. McDonald. *Handbook of biological statistics*, volume 2. sparky house publishing Baltimore, MD, 2009.

[99] N. Meng, M. Kim, and K. S. McKinley. LASE: Locating and applying systematic edits by learning from examples. In *Proceedings of the 35th International Conference on Software Engineering*, pages 502–511, 2013.

[100] A. Miller, Z. Cai, and S. Jha. Smart contracts and opportunities for formal methods. In *Proceedings of International Symposium on Leveraging Applications of Formal Methods*, pages 280–299, 2018.

[101] J. M. Morse. Data were saturated... 25(5):587–588, 2015.

[102] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, pages 1287–1293, 2016.

[103] H. Muccini, A. Di Francesco, and P. Esposito. Software testing of mobile applications: Challenges and future research directions. In *Proceedings of the 7th International Workshop on Automation of Software Test*, pages 29–35, 2012.

[104] G. Muller, Y. Padioleau, J. L. Lawall, and R. R. Hansen. Semantic patches considered helpful. *ACM SIGOPS Operating Systems Review*, 40(3):90–92, 2006.

[105] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[106] L. R. Ness and P. I. Fusch. Are we there yet? Data saturation in qualitative research. 20(9):1408–1416, 2015.

[107] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Statistical learning approach for mining API usage mappings for code migration. In *Proceedings of the 29th International Conference on Automated Software Engineering*, pages 457–468, 2014.

[108] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. *arXiv preprint arXiv:1802.06038*, 2018.

[109] T. Nipkow, L. C. Paulson, and M. Wenzel. Isabelle/HOL: A proof assistant for higher-order logic. In *Lecture Notes in Computer Science 2283*, 2002.

[110] A. D. Ong and D. J. Weiss. The impact of anonymity on responses to sensitive questions. *Journal of Applied Social Psychology*, 30(8):1691–1708, 2000.

[111] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. *ACM SIGOPS Operating Systems Review*, 42(4):247–260, 2008.

[112] R. M. Parizi, Amritraj, and A. Dehghantanha. Smart contract programming languages on blockchains: An empirical evaluation of usability and security. In *Proceedings of the 1st International Conference on Blockchain*, pages 75–91, 2018.

[113] R. M. Parizi, A. Dehghantanha, K.-K. R. Choo, and A. Singh. Empirical vulnerability analysis of automated smart contracts security testing on blockchains. *arXiv preprint arXiv:1809.02702*, 2018.

[114] D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Rosu. A formal verification tool for Ethereum VM bytecode. In *Proceedings of the 27th International Symposium on Foundations of Software Engineering*, pages 18–21, 2018.

[115] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Spoon: A library for implementing analyses and transformations of Java source code. *Software: Practice and Experience*, 46(9):1155–1179, 2016.

[116] J. Pettersson and R. Edström. Safer smart contracts through type-driven development. Master's thesis, Chalmers University of Technology And University Of Gothenburg, Department of Computer Science and Engineering, 2016.
[117] S. Porru, A. Pinna, M. Marchesi, and R. Tonelli. Blockchain-oriented software engineering: Challenges and new directions. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 169–171, 2017.
[118] T. Punter, M. Ciolkowski, B. Freimut, and I. John. Conducting on-line surveys in software engineering. In *Proceedings of the 2nd International Symposium on Empirical Software Engineering*, pages 80–88, 2003.
[119] R3. Corda documents. https://docs.corda.net/, 2018, R3 Limited.
[120] F. Rahman, S. Khatri, E. T. Barr, and P. Devanbu. Comparing static bug finders and statistical prediction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 424–434, 2014.
[121] G. Rosu. An overview of the K semantic framework. Technical report, 2010.
[122] C. Sánchez, G. Schneider, and M. Leucker. Reliable smart contracts: State-of-the-art, applications, challenges and future directions. In *Proceedings of International Symposium on Leveraging Applications of Formal Methods*, pages 275–279, 2018.
[123] D. C. Sánchez. Raziel: Private and verifiable smart contracts on blockchains. *arXiv preprint arXiv:1807.09484*, 2018.
[124] B. Saunders, J. Sim, T. Kingstone, S. Baker, J. Waterfield, B. Bartlam, H. Burroughs, and C. Jinks. Saturation in qualitative research: Exploring its conceptualization and operationalization. *Quality & quantity*, 52(4):1893–1907, 2018.
[125] F. Schrans, S. Eisenbach, and S. Drossopoulou. Writing safe smart contracts in Flint. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, pages 218–219, 2018.
[126] P. L. Seijas and S. Thompson. Marlowe: Financial contracts on blockchain. In *Proceedings of International Symposium on Leveraging Applications of Formal Methods*, pages 356–375, 2018.
[127] P. L. Seijas, S. J. Thompson, and D. McAdams. Scripting smart contracts for distributed ledger technology. *IACR Cryptology ePrint Archive*, 2016:1156, 2016.
[128] I. Sergey and A. Hobor. A concurrent perspective on smart contracts. In *Proceedings of the 21st International Conference on Financial Cryptography and Data Security*, pages 478–493, 2017.
[129] I. Sergey, A. Kumar, and A. Hobor. Temporal properties of smart contracts. In *Proceedings of International Symposium on Leveraging Applications of Formal Methods*, pages 323–338, 2018.
[130] A. Sharma, F. Thung, P. S. Kochhar, A. Sulistya, and D. Lo. Cataloging GitHub repositories. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, pages 314–319, 2017.
[131] D. Shepherd, P. Francis, D. Weintrop, D. Franklin, B. Li, and A. Afzal. [Engineering Paper] An IDE for easy programming of simple robotics tasks. In *Proceedings of the 18th International Working Conference on Source Code Analysis and Manipulation*, pages 209–214, 2018.
[132] O. Sheyner and J. Wing. Tools for generating and analyzing attack graphs. In *Proceedings of the 2nd International Symposium on Formal Methods for Components and Objects*, pages 344–371, 2003.
[133] F. Shull, J. Singer, and D. I. Sjøberg. *Guide to advanced empirical software engineering*. Springer, 2007.
[134] L. Singer, F. Figueira Filho, and M.-A. Storey. Software engineering at the speed of light: How developers stay current using twitter. In *Proceedings of the 36th International Conference on Software Engineering*, pages 211–221, 2014.
[135] D. Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.
[136] J. Stark. Making sense of blockchain smart contracts. *CoinDesk, Published on June*, 4, 2016.
[137] A. Strauss and J. M. Corbin. *Grounded theory in practice*. SAGE, 1997.
[138] C. Sun, V. Le, and Z. Su. Finding and analyzing compiler warning defects. In *Proceedings of the 38th International Conference on Software Engineering*, pages 203–213, 2016.
[139] M. Swan. *Blockchain: Blueprint for a new economy*. O'Reilly Media, 2015.
[140] N. Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.
[141] F. Thung, X.-B. D. Le, D. Lo, and J. Lawall. Recommending code changes for automatic backporting of Linux device drivers. In *Proceedings of the 32nd International Conference on Software Maintenance and Evolution*, pages 222–232, 2016.
[142] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. SmartCheck: Static analysis of Ethereum smart contracts. In *Proceedings of the 1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain@ICSE*, 2018.
[143] P. Tsankov. Security analysis of smart contracts in datalog. In *Proceedings of International Symposium on Leveraging Applications of Formal Methods*, pages 316–322, 2018.
[144] P. Tsankov, A. Dan, D. D. Cohen, A. Gervais, F. Buenzli, and M. Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 25th Conference on Computer and Communications Security*, 2018.
[145] A. Unterweger, F. Knirsch, C. Leixnering, and D. Engel. Lessons learned from implementing a privacy-preserving smart contract in Ethereum. In *Proceedings of the 9th IFIP International Conference on New Technologies, Mobility and Security*, pages 1–5, 2018.
[146] M. Valenta and P. Sandner. Comparison of Ethereum, Hyperledger Fabric and Corda. *[ebook] Frankfurt School, Blockchain Center*, 2017.
[147] N. Valliappan, S. Mirliaz, E. L. Vesga, and A. Russo. Towards adding variety to simplicity. In *Proceedings of International Symposium on Leveraging Applications of Formal Methods*, pages 414–431, 2018.
[148] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy. CCAligner: A token based large-gap clone detector. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1066–1077, 2018.
[149] X. Wang, H. Wu, W. Sun, and Y. Zhao. Towards generating cost-effective test-suite for Ethereum smart contract. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering*, pages 549–553, 2019.
[150] D. Weintrop, A. Afzal, J. Salac, P. Francis, B. Li, D. C. Shepherd, and D. Franklin. Evaluating CoBlox: A comparative study of robotics programming environments for adult novices. In *Proceedings of the 36th CHI Conference on Human Factors in Computing Systems*, page 366, 2018.
[151] M. Wohrer and U. Zdun. Smart contracts: Security patterns in the Ethereum ecosystem and Solidity. In *International Workshop on Blockchain Oriented Software Engineering@SANER*, pages 2–8, 2018.
[152] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32, 2014.
[153] H. Wu, X. Wang, J. Xu, W. Zou, L. Zhang, and Z. Chen. Mutation testing for Ethereum smart contract. *arXiv preprint arXiv:1908.03707*, 2019.
[154] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 23rd conference on computer and communications security*, pages 270–282, 2016.
[155] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: State-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.
[156] H. Zhong and Z. Su. Detecting API documentation errors. In *Proceedings of the 27th International Conference on Object Oriented Programming Systems Languages & Applications*, pages 803–816, 2013.
[157] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API mapping for language migration. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 195–204, 2010.
[158] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, and M. Bailey. Erays: Reverse engineering Ethereum's opaque smart contracts. In *Proceedings of the 27th USENIX Security Symposium*, pages 1371–1385, 2018.
[159] T. Zimmermann. Card-sorting: From text to themes. In *Perspectives on Data Science for Software Engineering*, pages 137–141. Elsevier, 2016.
[160] W. Zou, D. Lo, Z. Chen, X. Xia, Y. Feng, and B. Xu. How practitioners perceive automated bug report management techniques. *IEEE Transactions on Software Engineering*, 2018.