A Mixed-Methods Study of Security Practices of Smart Contract Developers

Tanusree Sharma, Zhixuan Zhou, Andrew Miller, and Yang Wang, *University of Illinois at Urbana Champaign*

https://www.usenix.org/conference/usenixsecurity23/presentation/sharma

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

A Mixed-Methods Study of Security Practices of Smart Contract Developers

Tanusree Sharma[1], Zhixuan Zhou[1], Andrew Miller[1], Yang Wang[1]
[1]University of Illinois at Urbana-Champaign
{tsharma6, zz78, soc1024, yvw}@illinois.edu

Abstract

Smart contracts are self-executing programs that run on blockchains (e.g., Ethereum). While security is a key concern for smart contracts, it is unclear how smart contract developers approach security. To help fill this research gap, we conducted a mixed-methods study of smart contract developers including interviews and a code review task with 29 developers and an online survey with 171 valid respondents. Our findings show various smart contract security perceptions and practices, including the usage of different tools and resources. Overall, the majority of our participants did not consider security as a priority in their smart contract development. In addition, the security vulnerability identification rates in our code review tasks were alarmingly low (often lower than 50%) across different vulnerabilities and regardless of our participants' years of experience in smart contract development. We discuss how future education and tools could better support developers in ensuring smart contract security.

1 Introduction

Blockchains are cryptographic platforms that can securely host applications and enable the transfer of digital assets in a decentralized manner. Smart contract blockchains like Ethereum are increasingly capable of supporting sophisticated computations (a.k.a., decentralized apps or dApps) [94]. Smart contracts are program scripts that define customized functions and rules during transactions and can run autonomously once deployed on a blockchain [46]. To support this unique form of computation, domain-specific programming languages, such as Solidity and Vyper, have been created to allow developers to write smart contracts.

A wide variety of industry applications in finance, healthcare, and energy have been rapidly exploring the use of blockchain technology and smart contracts to enable system transparency and traceability. Since deployed smart contracts can perform critical functions of holding a considerable amount of digital assets, tokens or currencies in circulation, they become a hotbed for attacks. According to DeFi Pulse [18], there is about $29.93B USD worth of total value locked (TVL) as of Sept 2022 controlled by deployed smart contracts in Decentralized Finance (DeFi) applications. While DeFi is a promising domain and has the potential to disrupt traditional financial systems by lowering the barriers for billions of people who do not have access to these services, there has been several security incidents in which digital tokens worth of millions of dollars have been stolen. In June 2016, vulnerabilities in the Maker DAO (decentralized autonomous organization) smart contract code were exploited to empty out more than two million Ether, which were worth 40 million USD [85]. This attack exploited the reentrancy vulnerability in the 'splitDAO' function of the code. The code was poorly designed, allowing for a regular call to be changed into a recursive call, resulting in multiple unauthorized withdrawals and depletion of the account.

More recently, since DeFi started skyrocketing in 2020, there has been a new wave of smart contract attacks that led to the loss of hundreds of millions of dollars in value (e.g., [81]). Clearly, security is critical for smart contracts, and various smart contract security resources and tools have been created. However, it is not clear how people who write smart contracts (we denote as smart contract developers) think about and approach security in smart contract projects. To help bridge this gap, we conducted an exploratory qualitative study consisting of a semi-structured interview and a smart contract code review task with 29 smart contract developers with diverse backgrounds. We conducted an online survey with 171 valid responses to explore common security practices in smart contract development, following the results of an interview.

Research questions. Specifically, our study aims to answer the following research questions:

- **RQ1:** How do smart contract developers ensure their smart contracts are secure against potential attacks?
- **RQ2:** How do smart contract developers conduct code reviews and whether they are able to identify common smart contract security vulnerabilities in the code?

Summary of findings. Our study findings show that our participants have a wide variety of smart contract security perceptions and practices, including various tools and resources they used. Most participants did not consider security as a top priority in smart contract development, mainly because (1) they felt they needed to ship products fast, (2) they forked code from popular projects, and (3) they relied on security audits. The security vulnerabilities we tested had fairly low detection rates even for the popular reentrancy issue, only less than half of all participants successfully identified it. Task success rate was generally positively correlated with the level of experience (years of experience in smart contract development). Our hierarchical task analysis [86] of code reviews suggests that relying solely on standard documentation, reference implementations, and security tools was often inadequate for identifying security issues, particularly for junior developers. Despite accessing these materials or utilizing a security tool, these participants still failed to identify security issues. Besides, developers, regardless of their years of experience, shared challenges with existing smart contract security tools.

Main contributions. Our work makes the following contributions: (1) our rich interview data offer novel results on the various security perceptions and practices of smart contract developers with diverse backgrounds; (2) results from our smart contract code review task sheds light on how smart contract developers actually examine smart contract code for security vulnerabilities; (3) results from our online survey further explore common security practices of smart contract developers and largely confirm our interview findings, and (4) we present implications for security education and tools to support developers in ensuring smart contract security.

2 Related Work

2.1 Blockchains and Smart Contracts

The financial industry is seen as a primary user scenario of the blockchain concept. Blockchain is known for its use in cryptocurrencies, but it also has the potential in improving financial services by providing a secure and efficient way of tracking ownership over a series of transactions over time. In traditional financial systems, intermediaries such as banks are responsible for recording transactions, which harbor the risk of errors. However, with blockchains, transaction information is distributed across a network of nodes. Once a transaction is verified and recorded by the network, it cannot be altered [47, 72]. That significantly reduces the risk of errors.

Bitcoin was the first blockchain application and a cryptocurrency that provides users with full control over their transactions without geographical constraints, eliminating the involvement of centralized authorities such as governments and banks [71]. There are multiple blockchain platforms that offer different smart contract capabilities. There are also many smart contract languages, such as Solidity, Serpent, and Vyper. Solidity [20] is the most popular for Ethereum. It enables developers to build decentralized applications with built-in economic functions in smart contracts [24]. Smart contracts are essentially containers of code that encode real-world contractual agreements in the digital realm [65]. It is a protocol that automatically verifies and executes the terms of the agreement between parties in a decentralized blockchain network. Its code execution is verified by the network nodes [24]. Many languages can be used to write smart contracts, but Solidity [20] is the most popular for Ethereum. Solidity code is compiled into bytecode for execution on the Ethereum Virtual Machine (EVM) [1]. The EVM is a machine that runs smart contracts on Ethereum, using an instruction set to identify and call different contracts, manage exceptions, and compute transaction costs. Data is stored on the blockchain or in contract memory, and contracts are often used to handle and transfer Ether. Since Ethereum is arguably the most popular smart contract platform and has a wealth of tools and resources, making it a good choice for our developer research.

2.2 Software Development Practices

Developers play a crucial role in software development and their skills and domain knowledge are important in creating secure applications. Previous studies have pointed out the need for a deeper understanding [22] of software development [78] and the human factors of software security [51], highlighting the importance of security education [73, 95] for developers and understanding the tools and attitudes of developers towards security [33]. To advocate software security education, some indicated the need for security guidelines mandated by the industries [93], [96] in cultivating expertise [27, 73] to identify vulnerabilities. Furthermore, tools producing false positives [39, 57], complex workflows [39, 87] and poorly designed warning prompts [39] failed to meet the expectation of developers. By soliciting input from academia, industry, and government agencies, some institutes such as NIST [17] and OWASP [16] developed systematic software security guidelines for developers to mitigate the potential impact of exploitation and address the root causes of vulnerabilities in preventing future recurrences. In contrast, prior research on smart contracts mainly focused on the effectiveness of security assessment tools [43] in identifying vulnerabilities [30, 68], shortcomings in security tooling (i.e., the user interfaces for visualizing results and error messages [44]), and root causes for the occurrence of severe smart contract vulnerabilities [63]. However, notably they mainly focused on technical issues. These approaches were helpful. However, they did not examine smart contract developers' actual development and security practices with the programming languages which could result in buggy code. A closer look at the incidents revealed that eventually, a large number of vulnerabilities in smart contracts occurred due to developers' mistakes [63]. Hence, to help avoid vulnerabilities in smart contracts, there needs to be a well-thought-out design adhering to security best practices [43] and incorporating developers' needs as a key requirement for new tooling design of smart contract [40]. In this work, we explored an untapped perspective, smart contract developers' security practices.

2.3 Smart Contract Development Practices

2.3.1 Characteristics of Smart Contract Development

Compared with traditional software, smart contract development is relatively new. As a piece of software, a smart contract is quite similar to traditional software, while the behavior of the attacker/ attacks can be qualitatively different. Smart contracts' prominent characteristics allow interaction without a lower-level security library (i.e., digital signature). Thus, make them tempting targets to monetize the attacks (e.g., "stealing" tokens and passing them to a mixer before selling them) [48]. Recent literature shows common vulnerabilities in EVM-based smart contracts, including re-entrancy [67,76,90], unhandled exception [31, 64], integer overflow [62, 76], and unrestricted action or access control [76, 83]. To trigger re-entrancy, a function call is made to an external contract which invokes and re-enters callback to the original contract [67]. Integer overflows could also cause contract funds to become completely frozen [76]. In addition, lack of authorization checks can lead to execution of arbitrary code [83].

Blockchain technologies are evolving dramatically, which adds *"design flaws in smart contract languages"*. Developers of decentralized apps are often confronted with changing platform features [55]. Thus, common software weaknesses such as access control, incorrect calculation, race condition, and many other security weaknesses may be amplified on blockchain platforms [19]. In a smart contract, the system states are visible on public blockchains and the smart contract code is often *"open-sourced."* Furthermore, the incorporation of economic considerations, such as incentive mechanisms and gas costs, as outlined by Parizi et.al. [74] adds an additional layer of complexity to the overall concept of economic security. With the recent boom of the DeFi industry (since 2020), a new pool of developers joined this industry and started writing smart contracts [37]. Recent blockchain applications indicated the practice of *"calling external smart contracts"* during development [37], which often leads to potential exploitation. In fact, DeFi projects are known as *"money lego"* because they can be created by composing smart contracts from different projects [37]. This composability, however, also opens the door for trusting and calling untested and potentially vulnerable smart contracts.

The infamous disasters involving the DAO [14] and the Parity Wallet [9] have highlighted such risks where attackers exploited programming bugs to steal approximately USD 70M. A series of suspicious transactions happened in yCREDIT and xToken smart contracts and suffered attacks with a loss of USD 24M due to the inconsistency in minted tokens [11]. Another attack was due to a flash loan within the xSNXa contract [11]. Recent work shows systemic consensus-layer vulnerabilities due to miner extractable value (MEV) where attackers can front-run orders by observing and placing their orders with higher gas fees [41].

2.3.2 Empirical Research of Smart Contract Developers

Prior work has proposed theoretical foundations for smart contract development, e.g., an engineering process model [84]. There are prior studies on smart contract development practices (e.g., challenges encountered [59, 69], usability of smart contract languages [75], maintenance methods after contract deployment [36], usage of selfdestruct function [35], listing smart contracts on etherscan [52], controlling gas consumption [32]). However, these studies did not focus on security.

There are a few prior studies that explored smart contract developers' security perceptions. Zou et al. explored the perceptions and challenges of smart contract development, where security was one type of challenges discovered through interviews and surveys [97]. The majority (84.9%) of their survey respondents agreed that code review is essential to ensure smart contract correctness. Wan et al. [92] focused on self-reported perceptions of smart contract security and how it fits into the development lifecycle. Chaliasos et al. conducted a survey of smart contract developers to understand how they use automated security tools [34]. Developers tended to employ lightweight tools. The majority of auditors (76%) spent only up to 20% of their time using security tools during audits, implying that audits are mostly manual [34].

To our knowledge, our study is one of the very first to empirically examine smart contract developers' security perceptions, practices, and *actual behavior* in code review (i.e., whether and how they detect security vulnerabilities in smart contracts). Despite various attacks reported and evaluations of security analysis tools [77, 82, 82], we still know little about whether, when, and how smart contract developers actually deal with the security aspect of their contract code. In this work, we aim to uncover smart contract developers' security and development practices and the fine-grained details of how they attempt code review with existing tools and resources.

2.3.3 Smart Contract Security Methods/Tools

Smart contract developers have adopted a wide range of tools, such as ConcenSys [2] to operate at different stages of smart contract development. Many tools have been developed to analyze either contract source code or its compiled EVM bytecode [77], most of which are based on symbolic execution [60]. Some exmaples smart contract analysis tools include Oyente [3], ZEUS [58], Maian [4], SmartCheck [88], ContractFuzzer [56], Vandal [31], Ethainter [30], Securify [90], and MadMax [50]. For instance, Oyente [3] is a symbolic execution tool to detect a small number of known smart contract vulnerabilities. In comparison, Securify [90] uses a dependency graph to extract precise semantic information to verify compliance and violation. However, Securify suffers compatibility issues when used on operating systems other than

Linux. A recent study [79] highlighted the limitations of security analysis tools, including the ability to identify a limited number of known vulnerabilities, slow analysis speed, and a lack of cross-platform compatibility.

Code auditing is indicated as an essential aspect of defensive programming, commonly used in smart contract development. Popular tools for auditing include Surya [5], Mythril [6], and MythX [7]. Organizations also hire third-party companies like Trail of Bits, OpenZeppelin, and ConsenSys Diligence. Auditing is an iterative process and depends on factors like time, budget, and resources. Research suggests that code auditing should be an additional security measure rather than a primary consideration and recommends a *"security from the beginning posture"* [26, 80]. Organizations also run bounty programs to incentivize community members to report vulnerabilities. For example, the 0x project [10] offers bounties of up to \$100K USD for critical vulnerabilities, however, it depends on budget and resource availability. In this study, we aim to investigate developers' current practices and desires for future smart contract security tooling.

3 Method

We used a mixed-methods approach to study smart contract developers' security practices via semi-structured interviews, code review tasks, and a follow-up online survey to triangulate their self-reported perceptions with their actual behavior.

3.1 Interview Study

Our study was inspired by user studies of software developers' security practices (e.g., [25, 49, 75]). We explored smart contract developers' practices and expectations for security tooling through a user study, which includes semi-structured interviews and a code review task. The focus is on Solidity developers, as it is the most popular smart contract language [42, 75]. The study was approved by the IRB, and each participant was compensated with a \$30 gift card. Data was collected and quotes were anonymized to protect participants' privacy. Most of the study was conducted in 2022.

3.1.1 Participant Recruitment

To help reach a wide range of developers, we recruited through different methods: (1) snowball sampling from our contacts in the Ethereum community, (2) posting on our Twitter and Facebook as well as *ethresear.ch* and Discord channels, and (3) contacting Solidity developers of public smart contract projects on GitHub. We selected participants based on the responses to our screening survey. Respondents were invited to our study if they met our selection criteria: a) is a solidity-based smart contract developer; b) has some smart contract development experience; c) should provide a proper explanation of their developed smart contract(s). We did two rounds of recruitment. In total, we received 67 responses from our screening survey. We reached out to 38 of them via email based on our selection criteria. In total, 29 people agreed to participate in our study. Eight participants were from word-of-mouth by our personal contacts in the Ethereum community. Two were from GitHub public projects. 19 were from our concurrent postings on Twitter, Facebook, Discord, and the ethresear.ch online forum (we did not know who came from which of these specific platforms though). We stopped recruiting because our last four interviews did not generate new results. Participation was voluntary. The study lasted one hour with the initial interview 25 minutes, the code review task 25 minutes, and the exit interview 10 minutes.

3.1.2 Pilot study

We conducted two rounds of pilots with a total of four smart contract developers to test our study design. We revised our interview questions and code review tasks based on their feedback. Details of the pilot study and the changes we made based on the pilot results can be found in Appendix A

3.1.3 Initial Interviews

We conducted an exploratory qualitative study with semi-structured interviews to gather rich data on smart contract developers' security practices, a new phenomenon. Interview scripts were designed based on research questions and can be found on our study GitHub [1]. We started by asking about their programming languages background, role, experiences in software development, and motivation for starting smart contract development. Then we asked a series of questions to understand their knowledge and experiences of developing smart contracts and handling security issues, including tools for writing contracts, guidelines for smart contract development, factors considered, and challenges encountered during development. To understand how smart contract developers tackle potential security risks in the development process, we also asked about their current practices (e.g., use of coding standards, policies, and security analysis tools as well as any educational resources) related to smart contract security. To get rich anecdotal data, we also asked about their personal experiences and stories of how they handled specific smart contract security-related issues in the past. Thus, interview questions get at not only general practices but also specific cases. The next study component was a smart contract code review task to show how "concretely" participants go about reviewing the code and identifying potential vulnerabilities.

3.1.4 Smart Contract Code Review Task

Since code review is a common task in smart contract development, this task was designed to understand how smart contract developers conduct code reviews, particularly for identifying security vulnerabilities. Each participant was asked to review one smart contract that we created. Specifically, we asked them to (1) share their computer screen and allow us to record

[1]Our study scripts, smart contracts for the code review task, and analysis code book are in: https://github.com/AccountProject/Developer_Study_SC

the screen with their permission to understand how they conduct the code review; (2) have at most 25 minutes for the code review; (3) search/use any resources/tools they need; (4) i. review the code; ii. identify security vulnerabilities and/or areas for improvement; iii. modify the code accordingly; (5) explain the modifications and rationale behind them.

Code Review Task Design. We designed this study component to model the real-world code review task that developers would be reasonably expected to encounter in their smart contract development. We chose to include two vulnerabilities in each smart contract, where one vulnerability is more well-known and should be easier to identify than the other. We measured the difficulty of these vulnerabilities based on our pilot participants' feedback. We also added some minor non-security issues, such as indentation, space/tabs, and blank lines, which should be easily detectable.

To select smart contracts vulnerabilities, we conducted an extensive review of exploited smart contracts from different security vulnerability reports, such as Smart Contract Weakness Classification and Test Cases (SWC) [19], Consensys Known Attacks [2], different GitHub repositories and etherscan source code of abandoned or previously exploited smart contracts [12] and recent literature on smart contract exploitation ([76, 82]). The four chosen vulnerabilities are 1) reentrancy, 2) unchecked low-level calls, 3) integer overflow and 4) improper access control. The first two vulnerabilities were included in one contract, and the last two vulnerabilities in another contract. The vulnerabilities were grouped as they normally appeared in smart contracts. They were designed based on real-life exploitation, which has happened before [19, 76, 82]. However, it is possible that some vulnerabilities were easier or harder to find depending on their functions/method structure. The contract code can be found in the GitHub repo. Table 2 summarizes the four vulnerabilities in the code review task and how to avoid/address them.

We created our two smart contracts using a boilerplate contract of ERC-20 token which provides basic functionalities to transfer tokens and allow tokens to be approved so they can be spent by another account. The ERC20 token contract is commonly used as a basis for smart contract (DeFi) projects. Numerous DeFi projects and open-source smart contracts adopt the ERC20 standard interface. The ERC20 standard interface can create tokens on Ethereum and can be re-used by applications such as wallets and decentralized exchanges. We believed that most smart contract developers would have some familiarity with this contract, and our pilot study confirmed our assumption. Our contracts for code review have the same basic ERC-20 functionalities. To make our code review task realistic for participants to complete in 25 minutes, we trimmed down some auxiliary functions from the selected contract sample code. Both of the contracts for code review are ERC-20 based, and there are no major differences in size. We then modified the contract by adding the vulnerabilities. We got feedback from solidity programmers and tested the contracts in our pilot. We included comments in the contract code to help participants understand the context of the contract functions, as well as explicit comments to non-standard ERC20 token functions (interface).

During the task, we provided the instructions verbally. After giving the instructions, we provided the GitHub link which contained the smart contract for review to the participants through Zoom chat. We provided some extra time for participants to set up their development environment before we started counting the 25-min task time. This set-up time which was not counted in the 25 minutes. Since the target participants are very busy, we limited our study to about one hour. Code review time (25 minutes) is based on the code length, which varies significantly in practice (10-1000 lines of code). It is a limitation but for practical reasons it was a trade-off between study time and participant's availability.

3.1.5 Exit Interview

Once the task was completed or the time ran out, we conducted an exit interview with the participants. We asked their opinions about the task they worked on and how they perceived the difficulty of the task. We also asked about their overall experience with the code review task and if they would like to share any other experiences of security practices in smart contract development. Finally, we asked about their desired features of security tools to help them improve smart contract security. We received the modified code by each participant via email. This data helped us determine whether they correctly identified the vulnerabilities.

3.1.6 Data Collection and Analysis

Study data was collected through Zoom audio/video/screen recordings with consent and analyzed using thematic analysis [29]. Data included interview responses, code review task outputs, think-aloud, and exit interview responses.

Qualitative Data analysis. Two researchers performed open coding independently on a sample of the data (20%). Then they met regularly to discuss and converged on a shared codebook before coding the remaining data. We calculated the inter-coder reliability in Cohen's Kappa, which was 0.94 and considered very good [66]. Our open coding followed an inductive analysis method to explore practices and behaviors toward smart contract development. Our codebook includes 46 codes. Then data and concepts that belonged together were grouped into sub-categories. Further abstraction of the data was performed by grouping sub-categories into generic categories and those into main categories. We then grouped related codes, organized them in high-level themes, and iterated this process to finally produce 18 themes to interpret the results of smart contract developers' security practices, security concerns, and individual experiences and challenges with tools and development methods. Some example themes are: priorities in smart contract development, use of information resources, development tools, smart contract security tools.

Assessing The Code Review Task Outcome. For the code review task, recorded videos of the participants performing the code review task were analyzed to measure the success rate, based on whether a participant correctly identified the security vulnerabilities in the smart contract. Prior to conducting the lab study, we created and verified the correct/secure solutions for each task. The general ideas for these solutions are described in Table 2. This ensured that we could verify whether participants successfully identify the vulnerabilities and provide a correct/secure answer or fix. We also paid attention to how they conducted the code review, e.g., whether and how they searched and used any resources/tools. Specifically, we conducted a hierarchical task analysis [86], a common Human-Computer Interaction (HCI) technique, to break down the process of how a participant performed the code review into detailed steps and to help identify how the processes of successful and failed code reviews differ. We further dissected the steps into sub-steps to understand how smart contract developers inspected individual elements (e.g., functions, events) during the code review tasks. This analytical process helped us identify potential indicators for successes or failures of code reviews (e.g., reasoning, steps, resources, and tools that participants used during the task). For example, some participants checked the task code against common ERC20 standard implementations (e.g., OpenZeppelin) to ensure correct syntax, while others specifically checked certain functions (e.g., "transfer," "transferfrom," "owner") for vulnerabilities. We also checked whether their suggestions or modified code could potentially fix the vulnerabilities.

3.2 Online Survey

We conducted a follow-up online survey based on results from interviews and code reviews to explore common security practices among smart contract developers. The survey aimed to further generalize and quantify commonalities found in the interviews with a larger sample. We used our personal contacts in the crypto space, social media and smart contract community forums to recruit respondents. Each respondent who finished the survey and provided valid response can join a lottery to win a $100 gift card (one card for every 10 valid responses). We used a lottery approach for participant compensation in the survey as literature suggests that it increases participation in online surveys [28, 53, 54, 61, 91]. The time frame for the survey study was July, 2022-August, 2022.

3.2.1 Survey Design

This survey had a total of 30 questions, including open-ended questions, multiple-choice questions, and a code review section. The survey started with questions about demographics and participants' current practices around smart contracts, such as information sources and tools used for smart contract development and security, and familiarity with smart contract vulnerabilities. These questions were based on our interview results (Section 4). Next, the survey included a code review section. We created five smart contract code snippets and randomly assigned one snippet to each respondent. We asked them to review the code and suggest any improvements and they can use any tools/resources as they normally do. Similar to the code review tasks in the interviews (Section 3.1.4), we designed these tasks to model real-world code reviews. We chose to include one vulnerability in each code snippet.

In the exit interviews (Section 3.1.5), some participants suggested us consider two frequent smart contract exploit scenarios (i.e., front-running [13] and flash loans [38]). In a typical **front-running** attack, transaction order is manipulated by selfish or malicious actors, who copy the original transaction and have their copy executed first by setting a higher gas price [13]. **Flash loans** are a financial mechanism that allows borrowing a large amount of crypto assets without collateral, but the borrower must return all funds in the same block or the borrowing is reversed. Attackers can use flash loans to manipulate the price of a crypto asset on one exchange and then buy/sell it on another for price differences (i.e., arbitrage) [38]. The final survey included five vulnerabilities: 1) re-entrancy, 2) integer overflow, 3) improper access control, 4) front-running, and 5) flash loans. Following the code review, we asked questions about how they performed the code review. For instance, we asked *"How would you rate the difficulty of this task? (1 - not difficult at all, 10 - most difficult)"*. We also asked what resources they used during the code review. Our pilot studies indicate the survey takes about 30 minutes to complete, participants were instructed that they can complete the survey within 24 hours of starting and we recorded progress, timestamps and clicks during the code review sessions.

3.2.2 Survey Data Analysis

We received more than 1,000 responses but the majority had poor quality. We manually applied a combination of quality control/filtering criteria: (1) incomplete responses (i.e., some participants started but never finished the survey; Qualtrics flagged these responses as FALSE); (2) clearly meaningless open-ended answers (e.g., unintelligible characters or random numbers or words, such as *"dfgdfg," ""Scorpion tail lion The secret of silver," "ERUETHY8O3I4Y3OLYHTO3T."*), (3) unreasonably short time to complete the survey (<=10 minutes), and (4) unreasonable short time to complete code review (<= 5 minutes). After the filtering, we had a total of 171 valid responses. The survey took an average of 26 minutes to complete. We computed descriptive statistics of the demographic data and smart contract practices (e.g., tools used).

We considered all data from different sources (interviews, surveys, code reviews) in the overall analysis. Specifically, we analyzed the interview and survey data in a similar way (thematic analysis) and analyzed the code-review tasks in terms of task performance and task behaviors. Following the best practices of mixed-methods research, we triangulated the data from all three methods in identifying crosscutting patterns and themes. For instance, task performance data told us the actual review behavior but it did not tell us why they had such behavior. The interviews and surveys helped us understand why they had such behavior.

4 Findings

In this section, we present our study results in diverse views and practices of smart contract security. We also discuss results from our lab study and survey in terms of the task success rate, time to identify security vulnerabilities, and use of different code review approaches. In general, we observed that a higher percentage of our participants (from both studies) who had longer experience in smart contract development (e.g., 3+ years of experience) were able to identify these common vulnerabilities than the participants with shorter experiences.

4.1 Participants

Phase 1: interviews/code review sessions, we had a total of 29 interviewees (24 male, 5 female). Table 1 summarizes our interviewees' demographics. Our interviewees included 14 DeFi practitioners, 3 freelance contract dev., 3 software dev., 8 students, and 1 professor. For **Phase 2: online survey**, we had a total of 171 valid responses for our online survey. 69% were male and 31% were female. 93% of them had at least some college education. In terms of their years of experience in smart contract development, 66.7% of them had +3 years, followed by 25.7% with 1-3 years and 7.6% with <1 year.

Our interviewees (n=29) and survey respondents (n=171) used resources including Solidity documentation, Google search, Stack Overflow, YouTube, and Ethereum Foundation documents. The most popular tools were Truffle, Remix, and Ganache. Reusing code from libraries like OpenZepplin was common. 76% of interviewees had Solidity code review experience while 72% of survey respondents reported having it. 13.5% of survey respondents had code review experience in other programming languages. More details on participant programming background are in Appendix C.

4.2 Perceptions of Smart Contract Security

4.2.1 Security was not a priority

The majority of participants (83% of interviewees and 61% of survey respondents) did not claim security as a top priority and cited three main reasons. First, they needed to ship products fast (especially for those DeFi projects) and thus security becomes secondary. Second, many smart contract projects forked other popular projects (e.g., Uniswap), which often have already been vetted by the community and thus are probably secure. Third, they often relied on security audits conducted by someone else internally or externally. For instance, P8 said "*If you're planning to do an audit anyway, it kind of makes sense from a business perspective to ship code and then run it through multiple audits, instead of having your internal team [...]review the security at the same time.*" However, this approach, driven by business priorities, seems to substitute the need for proactive and continuous security practices throughout the development life cycle. In turn, it may result in delayed identification of security vulnerabilities.

Only five interviewees (P3, P11, P14, P26, P27) considered security a priority in smart contract development and P3 was the only one with less than one year of experience in this group. Similarly, 39% of all survey respondents considered security a top priority but none of them had less than one year of experience. These results imply that our participants with longer experiences were more likely to consider security as a top priority than their less experienced peers. For those who treated security as a top priority, they mentioned different aspects of security. For instance, P11 said: "*security [is] the most important factor because you need to know the address in the blockchain to send and receive, [...] the address should be unique and it's secure and cannot be changed by someone else.*" Our interviewees and survey respondents frequently mentioned reentrancy and front-running. Other security vulnerabilities mentioned were integer overflow, access control, delegate calls, denial of services, and oracle attacks.

4.2.2 What makes smart contract security hard?

Many interviewees suggested that ensuring smart contract security is hard. Five interviewees pointed out the inherent limitations in Solidity language design for maintaining security. For instance, P25 noted the ambiguity of Solidity programming language where function definitions are not explicit and suggested adopting functions like, in Python. P19 spoke about another issue with Solidity: *"Contract work[s] like state machine when send a transaction. It only appears like state changes. But in regular program, you can differentiate read-only calls and state changes. Solidity can not do that."* He also added the difficulties of performing proper string and array manipulations due to a lack of direct language/library support. To fill this gap, this participant created his own assembler to map between Solidity instructions and machine code instructions. P10 spoke about challenges around testing strategies (i.e., formal verification) due to the complexity of smart contracts. One such example is composite smart contracts which need to perform tasks through the use of external calls and the execution of other smart contracts that might belong to other owners or companies [23]. P10 expressed the difficulties of using some of the common software testing methods, such as invariants, fuzzy testing, and formal verification.

4.3 Practices of Smart Contract Security

In our interviews, we asked our interviewees to explain their practices of ensuring smart contract security. We start by discussing the different security strategies they followed.

4.3.1 Smart contract security strategies

Our interviewees described three broad categories of strategies: (1) software engineering best practices, (2) common software testing techniques, and (3) specialized strategies.

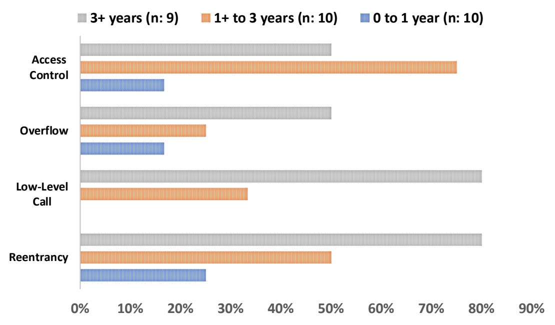


Figure 1: Interviewees' success rates of identifying security issues. The average success rates for senior, mid-level, and junior developers are 77.78%, 70% and 20% respectively.

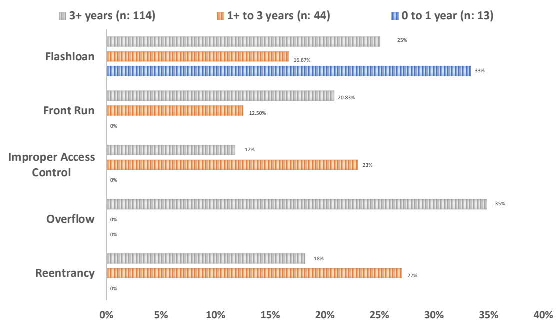


Figure 2: Survey Respondents' success rates of identifying security issues. The average success rates for mid-level and senior developers are 22.9% and 18.18%, respectively, while for junior developers, its 7.7%.

First, many interviewees talked about applying general software engineering best practices to help identify security issues. For instance, P20 emphasized the importance of code refactoring to "*write the most simple code that you can*" and this way makes the detection of security vulnerabilities much easier. Others mentioned having a modular structure and good documentation of smart contract code or drawing state machine diagrams to visualize the flow of smart contract code. In addition, using popular (vetted) libraries (e.g., Safe Math, OpenZeppelin libraries) and referring to official documentation (e.g., Solidity docs) were frequently mentioned as being helpful in avoiding code security problems. Furthermore, some interviewees said they chose to use more secure programming languages (e.g., Rust).

The second category of strategies was applying common software testing practices, such as code reviews, input validations, unit testing, and static analyses. However, some interviewees pointed out the difficulty of using certain testing techniques for complex smart contracts, such as formal verification (e.g., the example from P10 we presented earlier). Many interviewees also mentioned relying on security audits from external entities to ensure smart contract security.

The third category is specialized strategies that were created by developers because existing methods or techniques fell short. For instance, P18 explained that he created his own byte code (error code) dictionary to represent different cases of reverting transactions in his smart contracts for an NFT (non-fungible token) project. P23 discussed that his company doing its own simulations coupled with regular testing. These specialized strategies could be considered as potential features in future security tools. Next, we present their usage of security tools (and their limitations).

4.3.2 Use of smart contract security tools

While manual inspection of smart contract code was commonly reported, nine (31%) interviewees also reported using smart contract tools for security purposes, such as static analysis tools (17%), Truffle testing suite (14%), Remix security plugins (14%), MythX (7%) and Slither (3%). **Survey** results corroborate these interview results where manual inspection (64%) was frequently reported as the main method for ensuring smart contract security. Other tools were also mentioned, such as, Remix (37%), HardHat (36%), Slither (35%), and MythX (28%). We are not aware of other empirical data about usage of smart contract security tools. These results suggest that most participants relied on manual code inspection and for those who did use security tools they tended to use them as part of popular integrated development frameworks or environments for smart contracts (e.g., Truffle, HardHat, Remix) and to a less extent, seperate static analysis tools (e.g., Slither, MythX). This suggests that future security tools could be integrated into mainstream smart contract development stacks (e.g., Truffle, HardHat, Remix) for more adoption.

4.3.3 Limitations of smart contract security tools

76% of interviewees and 42% of survey respondents indicated the limitations of existing security tools for smart contracts and the needs for improving these tools.

One limitation is the lack of direct integration of automated contract testing into the development environments (IDEs). One concrete suggestion was to develop AI-based risk predictors and automatic vulnerability detection by leveraging existing vulnerability databases.

Another frequently mentioned limitation was that existing *symbolic execution* based tools, such as Slither and MythX in identifying edge cases that perform poorly for known vulnerabilities. Thus, they desired security libraries that provide more comprehensive coverage and higher success rates in identifying known vulnerabilities. A concrete recommendation frequently suggested was mathematical proof-based formal verification to enhance code coverage.

Importantly, many interviewees with many years of experiences noted that having better security tools alone is not enough and they advocated the need for having a *security* *posture* when writing smart contracts, "*security should be considered before you write your first line of smart contract code*." Education can be a way to improve such a security posture. After presenting how our participants' self-reported security practices, next we will see how they actually behaved in identifying security issues in the code review tasks.

4.4 Smart Contract Code Review Tasks

In this task, we asked both our **interviewees** and **survey respondents** to review a smart contract to identify any security issues or areas of improvement. We also asked their rationale for any code modifications.

4.4.1 Performance of identifying security vulnerabilities

15 of them were assigned to Task 1 (Reentrancy and Low-level calls), and the rest were assigned to Task 2 (Overflow, Access control) randomly. Overall, 55% of (16 out of 29) **interviewees** were able to identify one or more vulnerabilities during code review tasks. 28% (N=8) of interviewees identified both (all) vulnerabilities in the smart contract from which five were for Task 1 and three were for Task 2. More specifically, Reentrancy had the highest identification success rate (53%), followed by improper access control (43%), unchecked low-level call (40%), and lastly, integer overflow (29%). Average time in successfully identifying each security issue by interviewees: integer overflow took least time (average 8.6 minutes, median: 9.1, sd: 1.02), followed by low-level calls (average 9.8 minutes, median: 10.3, sd: 1.39), re-entrancy (average 11.6 minutes, median= 12.3, sd: 4.17), and access control (average 13 minutes, median: 14.1, sd: 2.65).

The code review results from our survey painted a similar high-level picture. A smaller percentage of **survey respondents** (20.5%, n: 171) identified vulnerability. More specifically, Integer overflow had the highest identification success rate (25.8%, n: 31), followed by flash loan (24.32%, n: 37), re-entrancy (19%, n: 37), front-running (17.64%, n: 34), and lastly unprotected Ether withdrawal relating to improper access control (15.6%, n: 32). As for participants' average time in successfully identifying each security issue: re-entrancy took the shortest (average 8 minutes, median: 7.9, sd: 1.67), followed by integer overflow (average 9.3 minutes, median: 8.8, sd: 2.59), front run (average 10 minutes, median: 8.8, sd: 3.35), improper access control (average 10.3 minutes, median: 8.6, sd:5.14), and lastly flash loan (average 12.2 minutes, median: 9.4, sd: 3.39). Table 2 in the Supplementary in GitHub summarizes the average time for successful identification.

4.4.2 Longer experiences in smart contract development positively associated with better task performance

To explore the relationship between developers' years of experience in smart contract development and their code review task performance, we first categorized our interviewees and survey respondents into different groups based on their years of experience. Year of experience is easy to understand and measure and is often used in smart contract job descriptions, which we used to come up with the thresholds (years of experience) for different groups. To our knowledge, no prior literature defines different levels of smart contract developers. Prior studies in software security have utilized years of experience as a quantitative proxy measure for expertise level but their thresholds for experienced developers vary. For instance, Acar et al. used having more than two years of experience for professional developers [22] whereas Witschey et al. [93] used six months for experienced developers.

We turned to smart contract job descriptions for inspiration. Specifically, we crawled Solidity smart contract job posts for developers from popular sites, including, Indeed (n=503), Glassdoor (n=74), cryptojoblist (n=58), and web3.career (n=88). We used keywords such as *"Solidity developer/engineer," "blockchain developer/engineer," and "Ethereum developer/engineer"* to search for the relevant jobs. For each of the websites, we crawled the job posts which were open and available (accessed on 18th November 2022), leading to a total of 723 job posts. We manually checked the job posts, deleted overlapped posts across the sites, and only kept jobs for Solidity developers during the data pre-processing, leaving a total of **155 smart contract job posts** from 75 crypto/blockchain organizations. The vast majority of these posts mentioned years of experience and there were three broad types of job posts: 1) Junior, 2) Mid-level, and 3) Senior/Lead. We found the following years of experience requirement for Junior (n=9, min: 0, max: 1 year, avg: 5.67 months, median: 6 months), Mid-level (n=77, min: 1, max: 3, avg: 1.93, sd: 0.70, median: 2), and Senior (n=69, min: 2, max: 5.5, avg: 3.57, sd: 1.07, median: 3.5). We used these empirical data to operationalize the thresholds of years of experience and put our participants in the following three groups: (Junior: 0 - 1year, Mid-level: more than 1 - 3 years, Senior: 3+ years).

Of the 29 interviewees, 16 identified at least one vulnerability, from which two were junior developers, seven were mid-level, and seven were senior developers. Senior developers had a higher task success rate (78%, n:9), followed by mid-level (70%, n:10) and lastly, junior (20%, n:10). For our survey responses (n=171), 35 identified vulnerabilities, from which one was a junior developer, eight were mid-level, and 26 were senior developers. Senior developers had a higher task success rate (23%, n:114), followed by mid-level (18%, n:44), and lastly, junior (7.6%, n:13). However, we have not found statistically significant results based on 2-sample proportion tests. Future research can further perform the analysis with a bigger sample proportion of each category of developers. To explore whether our high-level results are sensitive to the categorizations (cutoffs) of the years of experience, we also tried a few sets of cutoffs, for instance, junior (<1 year), mid-level (1-2 years) and senior (>2 years). We also tried the binary categorization of junior (<1 year) and experienced (>1 year). While the descriptive statistics had minor changes, the high-level results (e.g., positive association between years of experience and task success rate) were consistent. However, future research could explore additional measures of smart contract development experience and those relationships with a larger sample size for each developer category.

4.4.3 Different approaches to code review

In our interviewees, we were able to directly observe (via Zoom participant screen sharing upon consent) how our **interviewees** tried to identify security issues during the task. Most of them manually read through the code. Only four used security plugins and/or static analysis tools (the same set of security tools we presented in Section 4.3.2). For instance, P14 (had three years of experiences) used linting and static analysis tools in his code review. He successfully identified both vulnerabilities (reentrancy and unchecked low-level calls) in the contract. We did a hierarchical task analysis [86] to break down how a participant conducted the code review into detailed steps Figure 3 illustrates P14's code review process. P14 started by quickly setting up his development environment, where he created a folder with a package manager and installed development dependencies so he could compile the contract. He stressed the importance of having the folder ready to compile first to check coverage, test, and clean the code. After compilation, he utilized common implementations (e.g., OpenZeppelin) of the ERC20 standard to ensure the task code followed the correct syntax and used existing proprietary code to resolve identified vulnerabilities. Importantly, he did not blindly take or trust the Open Zeppelin implementation even though it is widely used. After manually checking the code, he explained some of the possible vulnerabilities. For instance, he pointed out a potential reentrancy vulnerability in the withdraw function: "*it is withdrawing if the amount is less than the amount to just return false and subtracts the amount before it does the accounting before it's sending anything out, which is pretty crucial for preventing someone re-entering the function, which would be bad.*" He then ran the static analysis with Slither to confirm his suspicion.

Unlike P14, many interviewees did manual code inspections without using any security tools. Some successfully identified the vulnerabilities while others did not. P4 (less than one year of experience) manually inspected the code without using any security tools and did not find any security vulnerabilities. He was assigned the same task (task 1) as P14. Figure 4 illustrates P4's code review process. He started to skim through the code, and preliminary commented there could be a bug and inconsistency to the ERC 20 standard. However, he didn't mention what is the exact bug in there. Then, he started to search topics on *"ERC20"* in google and spent much time understanding the functionality of the ERC20 token. Finally, he found Open Zeppelin ERC20 contracts for reference after spending much time searching. s the time was close to 25 minutes, he concluded - *"I definitely don't see a very critical issues at this point."* What is interesting here is that even though P4 found and read some reference documentation about ERC20 (e.g., Open Zeppelin) similar to P14, P4 was still unable to identify the security issues. In addition, many of our participants (e.g., P3, P15, P20, P21) who did use security tool(s) also missed some security issues.

4.4.4 Code modifications for improvement

For the 16 **interviewees** who have successfully identified at least one vulnerability, seven of them modified the contract code for improvement and six (21% of all participants) correctly fixed the vulnerabilities. The modified code snippets are included in the supplementary document. Due to time constraints, the other nine participants did not modify the code but verbally commented on how the contract code could be improved. Conceptually, all the comments would fix the security vulnerabilities. Of the 35 (20.5% of n=171) **survey respondents** who successfully identified the vulnerabilities, 31 provided suggestions on modifications. Conceptually, the comments would fix the security vulnerabilities. presents the improvement suggestions for Unprotected Ether Withdrawal and Front running. They explained how to implement controls so withdrawals can only be triggered by authorized parties while another respondent suggested to remedy by commit reveal hash scheme or to add a field to the inputs of approve function which is the expected current value to avoid front running. Detail can be found in supplementary materials.

4.4.5 False positives

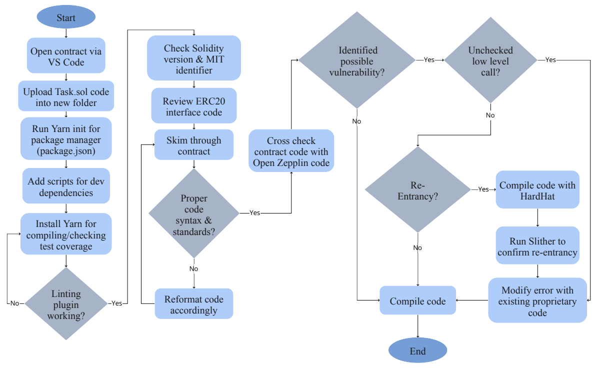
We also observed that sometimes our interviewees incorrectly thought they identified security vulnerabilities, which were actually not vulnerabilities (i.e., false positives). Four (14%) interviewees had false positives and they tended to have less prior experience with smart contract development. For instance, P1 thought there was a security issue in the "approve" function (Figure is in Supplementary document). He explained, *"it is possible for adversary to call the function in StandardToken and this would bypass 'whennotpaused' modifier."* However, there was a "Pausable" base contract, which can implement an emergency stop mechanism where "whenNotPaused" modifier was only callable when the contract is not paused. Also when it is called by the owner to pause, it triggers the stopped state. Hence, one cannot bypass the "whennotpaused" modifier.

5 Discussion

In this section, we unpack the implications of our key findings/observations for smart contract security as well as compare them to traditional software security practices.

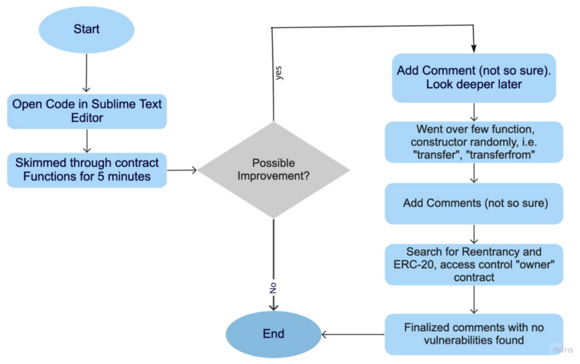
5.1 Summary of Major Findings

Overall, our interview and survey results suggest that a large portion of our participants did not consider security as a priority in their smart contract development. For the code review tasks, the vulnerability identification rate was also alarmingly low. While participants with longer experiences did better than their counterparts with shorter experiences, even the rate



Start
Open contract via VS Code
Upload Task.sol code into new folder
Run Yarn init for package manager (package.json)
Add scripts for dev dependencies
Install Yarn for compiling/checking test coverage
Linting plugin working?
No
Yes
Check Solidity version & MIT identifier
Review ERC20 interface code
Skim through contract
Proper code syntax & standards?
Yes
No
Reformat code accordingly
Cross check contract code with Open Zepplin code
Identified possible vulnerability?
Yes
No
Unchecked low level call?
Yes
No
Re-Entrancy?
Yes
No
Compile code with HardHat
Run Slither to confirm re-entrancy
Modify error with existing proprietary code
Compile code
End

Figure 3: P14's process for the code review task1. He used linting and static analysis tools and found vulnerabilities.



Start
Open Code in Sublime Text Editor
Skimmed through contract Functions for 5 minutes
Possible Improvement?
yes
No
Add Comment (not so sure). Look deeper later
Went over few function, constructor randomly, i.e. "transfer", "transferfrom"
Add Comments (not so sure)
Search for Reentrancy and ERC-20, access control "owner" contract
Finalized comments with no vulnerabilities found
End

Figure 4: P4's process for the code review task1. P2 failed to identify any security vulnerabilities.

of the most experienced group (those with +3 years) was concerning. We attributed these results largely to developers' lack of motivations, knowledge and skills of smart contract security, and the limitations of existing tools.

One reason that some participants did not consider security as a priority is the ideology of shipping products fast even at the cost of security. This ideology could have detrimental long-term effects on the smart contract ecosystem. How to counter this ideology? In addition to monetary loss and reputation cost as results of insufficient attention to security and associated security breaches, we hope that the market and users will gravitate towards those projects that treat security as a key goal. We also hope that the prominent projects can set good examples by explicitly discussing their system security practices in their official documents (e.g., white papers) so that there is an expectation for such information and in turn can encourage other projects to pay more attention to security.

A second reason is that our participants often reuse existing code libraries (e.g., Safe Math), which have often been vetted by the broader community. We believe this practice can be part of a promising approach, which is to have reference implementations that avoid or fix known security vulnerabilities (e.g., the Safe Math library fixes the underflow or overflow vulnerabilities) and then to educate developers about how to make proper use of these (vetted) libraries.

A third reason is that many participants relied on security audits often conducted by external entities. Many participants commented these external audits only occurred in the final stage of the smart contract development before the public project launch. These comments and observations suggest a misconception about security audits in the smart contract industry - instead of a continuous process, security audits were thought as a rather ad-hoc or proxy clearance to deploy the smart contracts or to ship the codes to various clients. In previous research on traditional software security, security audits are considered as a continuous process to keep track of infrastructure changes and to enforce rules accordingly [89]. The over-reliance on one-time external security audits could also have a negative long-term effect on the security and success of the smart contract ecosystem.

5.2 Smart Contract Security vs Traditional Software Security

Compared with traditional software, smart contract development is a relatively new landscape. With the recent boom of the DeFi industry (since 2020), a new pool of developers joined this industry and started writing smart contracts. Their knowledge of and experience with smart contract security vary widely, as observed in our study. There are some information resources on smart contract security, such as ConsenSys and Solidity documentation, which provide design patterns that developers should follow. However, we are not aware of a framework to assess and communicate smart contract security during the development life cycle.

Apply software security solutions in smart contracts. Our results suggest some similarities in practices and challenges between smart contracts and general software security. Thus some security strategies of software development can be applied to smart contracts. For instance, some participants indicated the importance of relevant documentation within the IDEs which can support their learning in smart contract development while software engineering has made similar suggestions (e.g., [21, 70]). Our participants also desired well-structured, context-specific security warnings. This aligns with prior literature that suggests effective security warning to reduce cryptographic API misuse [49]. The prior literature on cryptographic APIs [21] suggests that simplified libraries can promote security. However, it might not be the case in the domain of smart contracts. Many participants in our study explicitly requested the security/testing libraries to be more comprehensive in covering most if not all security issues. Further research is needed to investigate this idea.

Why is smart contract security different and difficult? Smart contract security have some fairly unique or more prominent characteristics and often have a significant financial impact. It is fairly easy to monetize the attacks (e.g., "stealing" tokens and passing them to a mixer before selling them). In addition, the system states are visible on public blockchains and the smart contract code is often open-sourced, thus the public nature makes them targets for attacks. Besides, the additional economic considerations (e.g., incentive mechanisms, gas cost) add more complexities to smart contract security (i.e., economic security) [74]. Furthermore, reusing library code or calling other smart contracts is common in smart contract development. In face, DeFi projects are known as "money lego" because they can be created by composing smart contracts from different projects [37]. This composability, however, also opens the door for trusting and calling untested and potentially vulnerable smart contracts.

5.3 Implications for Smart Contract Security

5.3.1 Education & Standards

Education. Our results suggest that accessing documentations, reference implementations and security tools is not enough in helping developers identify smart contract security vulnerabilities. Education that improves developers' motivation, knowledge and awareness regarding smart contract security is crucial. While there are many reading materials for educational purposes, we believe what is missing is hands-on exercises or labs for smart contract security, similar to the SEED projects for computer security in general [45]. Once these educational materials (e.g., hands-on labs) are created, they should be brought up when corresponding security issues are detected (i.e., teachable moments) in the various components of the smart contract ecosystem (e.g., compilers, security tools, IDEs, testnets).

In addition, the need for educational resources can differ depending on a developer's expertise/skill level. Smart contract education could be better designed, case by case basis and dispatched based on developers' self-reported expertise, skills, years of experience, and stages of development (i.e., writing code, reviewing code). An actual implementation could be a plugin or browser extension with a resource viewer, which can generate resources from various online and official sources upon the request sent by a developer. Such a plugin could be a part of existing IDE (i.e., Remix, Vscode) and security tools to support continuous referenced learning. This education tool can be trained to learn the evolution of developers' skills and information search by them. Thus, can suggest materials (i.e., updates of solidity language syntax, recent exploitation, why it happened, and way to resolve, related community discussion on mainstream media, etc.) from time to time.

Standardization. Survey respondents suggested standardizing smart contract security practices. For instance, while code audit is one of the main techniques for security assessment, they suggested to standardize the process of audit. Specially, organizations performing automated code audits should follow an standard agreement that covers the consumers during and after the audit especially if there is any exploitation. Our data also suggests an industry-wide misconception of audit where it is adopted in an ad-hoc manner before smart contract deployment rather than an iterative process. This could be due to resource (i.e., cost, SME) limitations for many industries. As a remedy, a security awareness program can be designed for the broader smart contract community to educate the consequences of an unsuccessful audit and what to expect from an audit. By nature, smart contract audit needs to be more frequent than existing software security audit. However, they can adopt the standard rules for baseline checks from well-established organizations, ISO/IEC, SOC2, etc.

5.3.2 System/Tool Design

Compilers. Our results indicate that most of our early-stage developer participants were not familiar with the basic smart contract security concepts and common vulnerabilities. Some suggested integrating security analyses (e.g., static analysis) directly into the Solidity compiler, so that they can obtain the security assessment without any extra step.

Code libraries. Some participants commented that current security libraries fall short of covering edge cases. Most of the recent tools work on the byte code for identifying security vulnerabilities. The parser and symbolic execution engine work solely on the byte code, so if developers see a potential integer under/overflow, they will report it, but they do not know where it occurs in the source code. A direct mapping between security vulnerabilities and source code would be valuable. Survey results confirmed the need of code libraries for symbolic execution, fuzzing, and linting.

Formal verification. In addition to static analysis, participants suggested integrating formal verification tools for smart contract security to enhance code coverage and correctness. Survey results also highlighted the importance of formal verification and the need for a formally verified specification of Solidity with better constructability and manageability.

Development frameworks. Some participants mentioned existing development frameworks to be heavy-weighted and difficult to learn and use the security functionalities. For instance, P6 noted Truffle, a popular smart contract development framework is "unwieldy" to learn and run security tests. One possibility is for these development frameworks to include a "security mode" where security analyses are automatically done as part of the compilation.

Improving existing tools. Survey respondents frequently expressed the need for improving existing smart contract security tools, mainly for the following purposes – functional correctness, fast/efficient assessment, more accurate bug detection, bug repair suggestions, and code optimizations.

5.3.3 User interfaces & user experience

Error / warning messages. Many participants found current testing library error messages hard to understand and lacking actionable insights. They suggested including links to detailed explanations, known incidents, and how to correct the issues. They also suggested having better GUI's to present the information, similar to that of other programming languages like Python. This would make security assessments more manageable and less overwhelming. Survey results also confirmed the need for better user experience and workflow in security tools by hierarchically zooming into where exactly the problems are in the code and how significant the effect can be.

Integrated Development Environments (IDEs). Our study suggested integrating security concepts into IDEs. A checklist of common smart contract security issues inside IDEs would be useful. The security tools (e.g., static analysis) can show which common security issues from the list are present in the code. Many participants liked the convenience of the web-based Remix IDE, but wished its features can match those in the desktop-based IDE (e.g., Visual Studio Code) where more security plugins are available. Some of them also desired the security plugins to have better discoverability through default enablement or better search tools."

5.4 Limitations and Future Research

Our mixed-methods study has many limitations.

First, while we had a large number of participants (29 interviewees, 171 survey respondents), we still cannot claim our results can necessarily be generalized to the broader smart contract developer population.

Second, our study focused on Solidity, the most popular programming language for smart contracts. There are other smart contract languages such as Viper and RUST. Further studies are needed for smart contract security practices in other languages. In addition, we only tested a few common smart contract vulnerabilities in our code review tasks. Future research is needed to investigate other smart contract vulnerabilities.

Third, the specific vulnerabilities and their groupings in the tasks we included might be perceived as having different levels of complexity, which could affect participants' code review performance. However, we believe this potential impact is rather limited for the following reasons. All the code review tasks were randomly assigned to participants (regardless of their experience level) and the code snippets were of similar length (around 80-100 lines of code excluding comments). In addition, the vulnerabilities were grouped as they have appeared in smart contracts. One such example is the DAO hack [8] where $low-levelcall$ function triggers the attacker contract's $fallback$ function, and that function tries to re-enter the *withdraw* function [19]. Furthermore, the smart contracts (ERC20) are frequently used by developers and are selected from real-life exploits with slight modifications [19, 76, 82]. Therefore, we believe the code examples in our tasks can reasonably represent actual smart contract code in practice. Furthermore, the time allocation for the code review task in the interview study was constrained to 25 minutes due to practical considerations, mainly the inability to schedule a session exceeding one hour given the participants' project-related pressures. We acknowledge that this methodological choice may limit the ecological validity and impact the task's success rate. In contrast, during Phase 2, during the survey, we instructed participants to complete the task within 24 hours to simulate practical scenarios, despite the survey's overall time was estimated as 30 minutes.

Fourth, our participants were asked about security practices in the interviews and the survey before doing the code review task. Therefore, that could prime our participants to think more about security in the code review and did better than they would otherwise. We also noticed a positive association between years of experience and successfully identifying security vulnerabilities in the code review task, this hypothesis however needs to be further tested in future studies.

Fifth, during survey data collection, we encountered challenges pertaining to data quality of 80% responses that were bogus. We employed a cautious approach involving extensive filtering criteria and manual inspection leading to a total of 171 responses. We are confident that these responses are legitimate due to the fact that the code review tasks and open-ended questions were designed to be answerable only by individuals knowledgeable in smart contracts. Nonetheless, we acknowledge the limitations inherent in our recruitment strategies and propose that future research endeavors focus on carefully selecting designated forums and leveraging personal contacts within the targeted community. This can mitigate the influx of spam or bot-generated responses that were obtained through social media platforms, particularly Twitter. Finally, we used our participants' self-reported years of smart contract development experience as a reasonable proxy for their smart contract expertise level to explore its relationship with their task performance. While years of experience is a reasonable metric and has been used in the prior developer studies, it is not the only useful metric. For instance, other reasonable metrics could include the number of smart contract projects worked on or smart contract languages used. Our research was not aimed to identify latent clusters of smart contract developers based on their knowledge and practices of smart contract development, which is nevertheless a valuable topic for future research.

6 Conclusion

To understand smart contract developers' security perceptions and practices, we conducted a mixed-methods study consisting of interviews, code review tasks and an online survey. The majority of our participants did not consider security as a priority and they often relied on external audits to ensure security of their projects. Given the recent rise of smart contract projects (e.g., decentralized finance) and their associated security attacks, providing better educational materials and tool support especially for developers is paramount for the healthy growth of this domain.

References

[1] Accessed on 2022. https://ethereum.org/en/developers/docs/evm/.

[2] Accessed on 2022. https://consensys.github.io/smart-contract-best-practices/known_attacks/.

[3] Accessed on 2022. https://github.com/enzymefinance/oyente.

[4] Accessed on 2022. https://github.com/ivicanikolicsg/MAIAN.

[5] Accessed on 2022. https://github.com/ConsenSys/surya.

[6] Accessed on 2022. https://github.com/ConsenSys/mythril.

[7] Accessed on 2022. https://github.com/trufflesuite.

[8] Accessed on 2022. https://blog.chain.link/reentrancy-attacks-and-the-dao-hack/.

[9] Blockchain-based venture capital fund hacked for $60 million, Accessed on 2022. https://fortune.com/2016/06/18/blockchain-vc-fund-hacked/4.

[10] Bug bounty., Accessed on 2022. https://0x.org/docs/guides.

[11] Deposit less, get more: ycredit attack details, Accessed on 2022. https://blocksecteam.medium.com/deposit-less-get-more-ycredit-attack-details-f589f71674c3.

[12] Etherscan, Accessed on 2022. https://etherscan.io/directory/Smart_Contracts.

[13] frontrun, Accessed on 2022. https://solidity-by-example.org/hacks/front-running/.

[14] An in-depth look at the parity multisig bug, Accessed on 2022. https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/.

[15] Openzepplin, Accessed on 2022. https://openzeppelin.com/contracts/.

[16] Owasp secure software development framework, Accessed on 2022. https://owasp.org/www-project-security-knowledge-framework/.

[17] Owasp security knowledge framework, Accessed on 2022. https://csrc.nist.gov/Projects/ssdf.

[18] pulse, Accessed on 2022. https://defipulse.com/.

[19] Smart contract weakness classification and test cases, Accessed on 2022. https://swcregistry.io/.

[20] Solidity, Accessed on 2022. https://github.com/protofire/solhint.

[21] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L Mazurek, and Christian Stransky. Comparing the usability of cryptographic apis. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 154–171. IEEE, 2017.

[22] Yasemin Acar, Sascha Fahl, and Michelle L Mazurek. You are not your developer, either: A research agenda for usable security and privacy research beyond end users. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 3–8. IEEE, 2016.

[23] Mouhamad Almakhour, Layth Sliman, Abed Ellatif Samhat, and Abdelhamid Mellouk. A formal verification approach for composite smart contracts security using fsm. *Journal of King Saud University-Computer and Information Sciences*, 2022.

[24] Andreas M Antonopoulos and Gavin Wood. *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.

[25] Hala Assal and Sonia Chiasson. Security in the software development lifecycle. In *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*, pages 281–296, 2018.

[26] Hala Assal and Sonia Chiasson. 'think secure from the beginning' a survey with software developers. In *Proceedings of the 2019 CHI conference on human factors in computing systems*, pages 1–13, 2019.

[27] Dejan Baca, Kai Petersen, Bengt Carlsson, and Lars Lundberg. Static code analysis to detect software security vulnerabilities-does experience matter? In *2009 International Conference on Availability, Reliability and Security*, pages 804–810. IEEE, 2009.

[28] Michael Bosnjak and Tracy L Tuten. Prepaid and promised incentives in web surveys: An experiment. *Social science computer review*, 21(2):208–217, 2003.

[29] Richard E. Boyatzis. *Transforming Qualitative Information: Thematic Analysis and Code Development*. SAGE, April 1998.

[30] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 454–469, 2020.

[31] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981*, 2018.

[32] G. Canfora, A. Di Sorbo, S. Laudanna, A. Vacca, and C. A. Visaggio. Profiling gas leaks in solidity smart contracts. *arXiv preprint arXiv:2008.05449*, 2020.

[33] Justin Cappos, Yanyan Zhuang, Daniela Oliveira, Marissa Rosenthal, and Kuo-Chuan Yeh. Vulnerabilities as blind spots in developer's heuristic-based decision-making processes. In *Proceedings of the 2014 New Security Paradigms Workshop*, pages 53–62, 2014.

[34] S. Chaliasos, M. A. Charalambous, L. Zhou, R. Galanopoulou, A. Gervais, D. Mitropoulos, and B. Livshits. Smart contract and defi security: Insights from tool evaluations and practitioner surveys. *arXiv preprint arXiv:2304.02981.*, 2023.

[35] J. Chen, X. Xia, D. Lo, and J Grundy. Why do smart contracts self-destruct? investigating the selfdestruct function on ethereum. *ACM Transactions on Software Engineering and Methodology*, 2021.

[36] J. Chen, X. Xia, D. Lo, J. Grundy, and X. Yang. Maintenance-related concerns for post-deployed ethereum smart contract development: issues, techniques, and future challenges. *Empirical Software Engineering*, 2021.

[37] Xiangping Chen, Peiyong Liao, Yixin Zhang, Yuan Huang, and Zibin Zheng. Understanding code reuse in smart contracts. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 470–479. IEEE, 2021.

[38] Zhiyang Chen, Sidi Mohamed Beillahi, and Fan Long. Flashsyn: Flash loan attack synthesis via counter example driven approximation. *arXiv preprint arXiv:2206.10708*, 2022.

[39] Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 332–343, 2016.

[40] Michael Coblenz, Joshua Sunshine, Jonathan Aldrich, and Brad A Myers. Smarter smart contract development tools. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 48–51. IEEE, 2019.

[41] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 910–927. IEEE, 2020.

[42] Chris Dannen. Bridging the blockchain knowledge gap. In *Introducing Ethereum and solidity*, pages 1–20. Springer, 2017.

[43] Giuseppe Destefanis, Michele Marchesi, Marco Ortu, Roberto Tonelli, Andrea Bracciali, and Robert Hierons. Smart contracts vulnerabilities: a call for blockchain software engineering? In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 19–25. IEEE, 2018.

[44] Ardit Dika. Ethereum smart contracts: Security vulnerabilities and security tools. Master's thesis, NTNU, 2017.

[45] Wenliang Du. The SEED project: Providing hands-on lab exercises for computer security education. In *IEEE Security and Privacy Magazine, September/October*, 2011.

[46] Wesley Egbertsen, Gerdinand Hardeman, Maarten van den Hoven, Gert van der Kolk, and Arthur van Rijsewijk. Replacing paper contracts with ethereum smart contracts. *Semantic Scholar*, 35, 2016.

[47] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol with chains of variable difficulty. In *Annual International Cryptology Conference*, pages 291–323. Springer, 2017.

[48] David Gerard. *Attack of the 50 foot blockchain: Bitcoin, blockchain, Ethereum & smart contracts*. David Gerard, 2017.

[49] Peter Leo Gorski, Luigi Lo Iacono, Dominik Wermke, Christian Stransky, Sebastian Möller, Yasemin Acar, and Sascha Fahl. Developers deserve security warnings, too: On the effect of integrated security advice on cryptographic {API} misuse. In *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*, pages 265–281, 2018.

[50] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–27, 2018.

[51] Matthew Green and Matthew Smith. Developers are not the enemy!: The need for usable security apis. *IEEE Security & Privacy*, 14(5):40–46, 2016.

[52] P. Hartel, I. Homoliak, and D. Reijsbergen. An empirical study into the success of listed smart contracts in ethereum. *IEEE Access*, 2019.

[53] Dirk Heerwegh. An investigation of the effect of lotteries on web survey response rates. *Field Methods*, 18(2):205–220, 2006.

[54] Gary Hsieh and Rafał Kocielnik. You get who you pay for: The impact of incentives on participation bias. In *Proceedings of the 19th ACM conference on computer-supported cooperative work & social computing*, pages 823–835, 2016.

[55] Yongfeng Huang, Yiyang Bian, Renpu Li, J Leon Zhao, and Peizhong Shi. Smart contract security: A software lifecycle perspective. *IEEE Access*, 7:150184–150202, 2019.

[56] Bo Jiang, Ye Liu, and WK Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 259–269. IEEE, 2018.

[57] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

[58] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In *Ndss*, pages 1–12, 2018.

[59] N. Kannengießer, S. Lins, C. Sander, K. Winter, H. Frey, and A. Sunyaev. Challenges and common solutions in smart contract development. *IEEE Transactions on Software Engineering*, 2021.

[60] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[61] Jerold S Laguilles, Elizabeth A Williams, and Daniel B Saunders. Can lottery incentives boost web survey response rates? findings from four experiments. *Research in Higher Education*, 52(5):537–553, 2011.

[62] Enmei Lai and Wenjun Luo. Static analysis of integer overflow of smart contracts in ethereum. In *Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy*, pages 110–115, 2020.

[63] JIANG P LI XQ, T CHEN, et al. A survey on the security of blockchain systems. *Future Generation Computer Systems*, 2018.

[64] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269, 2016.

[65] Daniel Macrinici, Cristian Cartofeanu, and Shang Gao. Smart contract applications within blockchain technology: A systematic mapping study. *Telematics and Informatics*, 35(8):2337–2354, 2018.

[66] Mary L McHugh. Interrater reliability: the kappa statistic. *Biochemia medica*, 22(3):276–282, 2012.

[67] Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M Kim, and Marek Laskowski. Understanding a revolutionary and flawed grand experiment in blockchain: the dao attack. *Journal of Cases on Information Technology (JCIT)*, 21(1):19–32, 2019.

[68] Alexander Mense and Markus Flatscher. Security vulnerabilities in ethereum smart contracts. In *Proceedings of the 20th International Conference on Information Integration and Web-Based Applications & Services*, pages 375–380, 2018.

[69] M. Möhring, B. Keller, R. Schmidt, A. L. Rippin, J. Schulz, and K. Brückner. Empirical insights in the current development of smart contracts. *PACIS*, 2018.

[70] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, Marco Herzog, Sergej Dechand, and Matthew Smith. Why do developers get password storage wrong? a qualitative usability study. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 311–328, 2017.

[71] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Satoshi Nakamoto Institute*, 2008.

[72] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. manubot. *Tech. Rep.*, 2019.

[73] Daniela Oliveira, Marissa Rosenthal, Nicole Morin, Kuo-Chuan Yeh, Justin Cappos, and Yanyan Zhuang. It's the psychology stupid: how heuristics explain software vulnerabilities and how priming can illuminate developer's blind spots. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 296–305, 2014.

[74] Reza M. Parizi, Amritraj, and Ali Dehghantanha. Smart Contract Programming Languages on Blockchains: An Empirical Evaluation of Usability and Security. In Shiping Chen, Harry Wang, and Liang-Jie Zhang, editors, *Blockchain – ICBC 2018*, Lecture Notes in Computer Science, pages 75–91, Cham, 2018. Springer International Publishing.

[75] Reza M Parizi, Ali Dehghantanha, et al. Smart contract programming languages on blockchains: An empirical evaluation of usability and security. In *International Conference on Blockchain*, pages 75–91. Springer, 2018.

[76] Daniel Perez and Ben Livshits. Smart contract vulnerabilities: Vulnerable does not imply exploited. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[77] Daniel Perez and Benjamin Livshits. Smart contract vulnerabilities: Does anyone care. *arXiv preprint arXiv:1902.06710*, pages 1–15, 2019.

[78] Olgierd Pieczul, Simon Foley, and Mary Ellen Zurko. Developer-centered security and the symmetry of ignorance. In *Proceedings of the 2017 New Security Paradigms Workshop*, pages 46–56, 2017.

[79] Purathani Praitheeshan, Lei Pan, Jiangshan Yu, Joseph Liu, and Robin Doss. Security analysis methods on ethereum smart contract vulnerabilities: a survey. *arXiv preprint arXiv:1908.08605*, 2019.

[80] Petri Puhakainen and Mikko Siponen. Improving employees' compliance through information systems security training: an action research study. *MIS quarterly*, pages 757–778, 2010.

[81] Jamie Redman. Flash Loan Attacks Drain 2 Binance Smart Chain Defi Projects for $6 Million, May 2021.

[82] Sarwar Sayeed, Hector Marco-Gisbert, and Tom Caira. Smart contract: Attacks and protections. *IEEE Access*, 8:24416–24427, 2020.

[83] Jinshan Shi, Ru Li, and Wenhan Hou. A mechanism to resolve the unauthorized access vulnerability caused by permission delegation in blockchain-based access control. *IEEE Access*, 8:156027–156042, 2020.

[84] C. Sillaber, B. Waltl, H. Treiblmaier, U. Gallersdörfer, and M. Felderer. Laying the foundation for smart contract development: an integrated engineering process model. *Information Systems and e-Business Management*, 2021.

[85] Emin Gün Sirer. Thoughts on the dao hack. *Retrieved February*, 18:2020, 2016.

[86] Neville A Stanton. Hierarchical task analysis: Developments, applications, and extensions. *Applied ergonomics*, 37(1):55–79, 2006.

[87] Tyler W Thomas, Heather Lipford, Bill Chu, Justin Smith, and Emerson Murphy-Hill. What questions remain? an examination of how developers understand an interactive static analysis tool. In *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*, 2016.

[88] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 9–16, 2018.

[89] Kennedy A Torkura, Muhammad IH Sukmana, Feng Cheng, and Christoph Meinel. Continuous auditing and threat detection in multi-cloud infrastructure. *Computers & Security*, 102:102124, 2021.

[90] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82, 2018.

[91] Tracy L Tuten, Mirta Galesic, and Michael Bosnjak. Effects of immediate versus delayed notification of prize draw results on response behavior in web surveys: An experiment. *Social Science Computer Review*, 22(3):377–384, 2004.

[92] Zhiyuan Wan, Xin Xia, David Lo, Jiachi Chen, Xiapu Luo, and Xiaohu Yang. Smart contract security: a practitioners' perspective. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1410–1422. IEEE, 2021.

[93] Jim Witschey, Shundan Xiao, and Emerson Murphy-Hill. Technical and personal factors influencing developers' adoption of security tools. In *Proceedings of the 2014 ACM Workshop on Security Information Workers*, pages 23–26, 2014.

[94] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[95] Glenn Wurster and Paul C Van Oorschot. The developer is the enemy. In *Proceedings of the 2008 New Security Paradigms Workshop*, pages 89–97, 2008.

[96] Jing Xie, Heather Richter Lipford, and Bill Chu. Why do programmers make security errors? In *2011 IEEE symposium on visual languages and human-centric computing (VL/HCC)*, pages 161–164. IEEE, 2011.

[97] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach Dinh Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering*, 47(10):2084–2106, 2019.

A Pilot Study Results

We conducted a pilot study with 4 smart contract developers to test our study design including interview questions and code review tasks. For the code review task, first we implemented five smart contracts, each including one common smart contract security vulnerability (e.g., reentrancy, under/overflow, access control). We present details of these smart contracts in Section 3.1.4.

First round pilot. In the first two pilot sessions, we had participants who were doctoral student researchers in the blockchain and smart contract security area. For second part of study (code review tasks), we asked pilot participants to review three contracts (10 minutes for each task), but they suggested that reviewing one contract with common functionalities would be more realistic and comprehensive. They also noted that the time allowed for review was too tight and that code review takes effort and can be distracting without a realistic contract. Therefore, we created two basic smart contracts based on ERC20 token standard, which have basic functionalities to transfer tokens as well as allow tokens to be approved so they can be spent by another on-chain party. ERC20 is the most common token standard on Ethereum and should be familiar to smart contract developers. The length of smart contract code varies significantly: some library/interface code can be 20 lines, while other more complicated contracts (e.g., Uniswap router code) can be several hundreds of lines. Practically, to fit into the time frame of our study, the code cannot be too long. Therefore, our 2 newly created smart contracts contain 80 lines of code on average. We chose an average of 80 lines of code based on this 1st round of pilot, which was sufficient to contain common vulnerabilities in the ERC20 form. We then embedded at least two common smart contract security vulnerabilities in each contract.

Second round pilot. To test the updated study materials, we had a second round of pilot with another two participants who were blockchain researchers/developers. The estimated time for the interview session was 30 minutes and 25 minutes for code review tasks and 5 minutes for exit interview. Specifically, we gave them 20 minutes to review the code (each participant was assigned only one task). They were asked to: a) review the contract code; b) update the code if necessary; c) can search for any resources online during the task. In addition, we gave them 5 minutes to discuss the area of improvement and their rationale after the task. They made suggestions on the interview questions. For instance, they suggested that we add questions to understand what role(s) the participant played in their smart contract projects. Therefore, we added questions to learn: a) what types of role they play in smart contract projects, and b) if they have any experience in deploying smart contracts in a production system or mainnet/testnet. They thought the smart contracts for review were good, but suggested making them ready to go compile from GitHub, which we did in the final study.

B Participants

Phase 1: interviews/code review sessions. We had a total of 29 interviewees (24 male, 5 female). Table 1 summarizes our interviewee demographics. More than one-third of our interviewees were from the US, and the rest were from many other countries, including India, China, Australia, Ghana, Egypt, Iran, UK, Canada, Germany, New Zealand, and Greece. They all had experience with smart contract development albeit with different years of experience. Specifically, 10 interviewees had less than one year of experience, another 10 had 1-3 years of experience, and nine had +3 years of experience. Our interviewees included 14 full-time DeFi smart contract practitioners, three freelance smart contract developers, three software developers mainly worked in other domains, eight college/graduate students, and one professor.

Phase 2: online survey. We had a total of 171 valid responses for our online survey. 69% were male and 31% were female. 93% of them had at least some college education. In terms of their years of experiences in smart contract development, 66.7% of them had +3 years, followed by 25.7% with 1-3 years and 7.6% with up to one year. The majority of them (79%) worked full-time in the crypto/blockchain industry. They all had experience with smart contract development but their main role varied. 44% was employed mainly for smart contract protocol development, followed by 35% in smart contract development, and 13% in smart contract research and security assessment.

C Programming & Development Background

Information resources used for smart contract development. Our **interviewees** (n=29) mentioned a number of information resources that they used for smart contract development. Common sources are Solidity documentation (34%), Google search (31%), Stack Overflow (17%), YouTube (14%), as well as documents published by the Ethereum Foundation (14%), OpenZeppelin [15] (14%), and ConsenSys [2] (14%). Our **survey respondents** (n=171) reported using similar resources such as, Google search (55%), Solidity documentation (43%), and OpenZeppelin (32%).

Tools for smart contract development. Our **interviewees** mentioned several tools that they used to develop, deploy, and test their smart contracts. Most frequently mentioned tools included: Truffle (76%), Remix (55%), Ganache (38%), HardHat(34%), and Waffle (21%). Our **survey** results were largely similar: Remix (43%), VS Code (38%), HardHat (35%), Ganache (34%), Geth (28%), and Truffle (17%). . Remix was one of the top tools in both studies.

Reusing other project code. Using existing code is a common practice mentioned by our interview and survey participants. 52% (15) of our **interviewees** with various years of experiences talked about using code from open source libraries such as OpenZepplin, a popular implementation of the ERC-20 and other ERC standards. 38% of our **survey respondents** reported using existing library code such as OpenZepplin, the Smartdev-contract library, Solmate, ethers-rs, ZKcontract and ConsenSys libraries.

Experience in code review. 76% of our **interviewees** had Solidity code review experiences. Some conducted code review by pen and paper (P1, P6, P20, P25), others used automated test scripts/tools (e.g., OpenZepplin ERC20 library) for code review (P2, P24, P26, P27), and some used both techniques (P14, P19, P24, P26). Among the seven interviewees who did not have code review experience in Solidity, three of them had code review experience in other languages, such as Python (P3), RUST (P8) and Golang (P22). Four interviewees (P4, P5, P13, P17) did not have any prior code review experience in any language. 72% of **survey respondents** did Solidity code review before. About 13.5% of all survey respondents mentioned having code review experience in other languages, including, Python, Rust, and Java.

Takeaways 1: Most participants mentioned using code from existing open-source libraries for smart contract development and conducting smart contract code reviews.

D Tables

ID	*Gender*	*Country*	*Occupation*	*Year(s) Exp.*
P1	Male	USA	Developer, DeFi Company	<1
P2	Female	USA	Developer, DeFi Company	2
P3	Male	USA	Masters Student, CSE	<1
P4	Male	Australia	Researcher, DeFi Company	<1
P5	Male	India	Bachelors student, CSE	<1
P6	Male	USA	PhD Student, CSE	5+
P7	Female	USA	Masters Student, CSE	<1
P8	Male	Ghana	Freelance Smart Contract Dev	4
P9	Male	USA	Professor, CSE	<1
P10	Male	USA	Developer, DeFi Company	4+
P11	Female	China	Dev, DeFi Company	2
P12	Female	Egypt	Freelance Smart Contract Dev	4.5
P13	Male	Iran	Software Developer	4
P14	Male	UK	Co-Founder,dev, DeFi Company	3
P15	Male	India	Software Developer	2
P16	Male	India	Freelance Smart Contract Dev	<1
P17	Male	India	Software Developer	2
P18	Male	USA	Dev, Defi Company	2.5
P19	Male	USA	CTO, DeFi Company	3+
P20	Male	USA	Bachelor Student, CSE	<1
P21	Male	India	Bachelor Student	2.5
P22	Male	Germany	PhD Student, CSE	3.5
P23	Male	Australia	Developer, DeFi Industry	2
P24	Female	Canada	PhD Student, CSE	<1
P25	Male	Greece	Developer and Researcher	3
P26	Male	Germany	Developer, DeFi Industry	3
P27	Male	Canada	Developer, DeFi Industry	4+
P28	Male	USA	Developer, De-Fi Industry	<1
P29	Male	New Zealand	Developer, DeFi Industry	4

Table 1: Participant demographics and background.

Table 2: Vulnerabilities in code review tasks as well as effective strategies to address these vulnerabilities. The 1st-4th vulnerabilities were used in the interview study, and the 5-6th vulnerabilities were added in the online survey (see details in Section 3.2.1).

Vulnerability	Description	Possible Prevention Technique
1. Reentrancy	Calling external contracts is that they can take over the control flow, and make changes to the data that the calling function is not expecting.	Make sure it does not call an external function or use the Checks-Effects-Interactions pattern.
2. Unchecked Low-Level Calls	This can lead to unexpected behaviour and break the program logic. A failed call can even be caused by an attacker, who may be able to further exploit the application	Ensure to handle the possibility that the call will fail by checking the return value.
3. Integer Overflow	An integer overflow occurs when an arithmetic operation attempts to create a numeric value that is outside of the range	Use of vetted safe math libraries for arithmetic operations consistently throughout the smart contract system.
4. Improper Access Control	Exposing initialization functions by wrongly naming a function intended to be a constructor, the constructor code ends up in the runtime byte code and can be called by anyone to re-initialize the contract.	Implement controls so withdrawals can only be triggered by authorized parties or according to the specs of the smart contract system.
5. Front-running	In a decentralized exchange where a buy order transaction can be seen, and a second buy order by bad actors (e.g., malicious miners) with higher gas price can be executed before the first transaction. The buy price is now higher for the original buyer.	Remove the benefit of front-running, use a pre-commit scheme, specify an acceptable price range on a trade
6. Flash loans	Bad actors take out a flash loan from a decentralized lending protocol, uses the borrowed funds to manipulate the price of a crypto asset on one exchange, and then make huge profits by transacting the crypto asset on another exchange if they have a price difference (i.e., arbitrage opportunities).	Limit the amount that can be borrowed in a single flash loan, use a time delay in price oracle or a pre-commit scheme in decentralized exchanges or lending protocols