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

ACM Reference Format:
. 2023. Supplementary Materials A Mixed-Methods Study of Security Practices of Smart Contract Developers. 1, 1 (February 2023), 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

This document supplements Sections 3 and 4of the main paper. In particular, it includes the following:

- Supplements Section 3
  (a) Pilot Study Details (referred in Section 3.1.2 in main paper)
- Supplements Section 4
  (a) Interviewees Programming background by Developers (referred in Section 4.1)
  (b) Survey respondents' use of Open Source libraries/ code/standards (referred in Section 4.1)
  (c) Interviewees Perceptions of Smart Contract Security: Factors considered in smart contract development by Developers (referred in Section 4.2)
  (d) Survey respondents' awareness of security vulnerabilities (referred in Section 4.2)
  (d) Findings from The Smart Contract Code Review Tasks: Task performance of identifying security vulnerabilities for Survey respondents (referred in Section 4.4)
  (e) Findings from The Smart Contract Code Review Tasks: Different approaches to code review (referred in Section 4.4)
  (e) Findings from The Smart Contract Code Review Tasks: Code modifications for improvement (referred in Section 4.4)
    a. Code Snippets from Interview participants
    b. Code snippets from survey respondents
- Supplements Section 3
  (a) Study Protocol (Interview)
  (b) Study Protocol (Code Review Sessions)
  (c) Study Protocol (Survey)
  (d) Recruitment Script

1 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

Author's address:

Manuscript submitted to ACM

smart contract security vulnerability (e.g., reentrancy, under/overflow, access control). We present details of these smart contracts in Section 3.1.5.

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. We conducted the interview and provided three contracts to each participant for review. They felt the interview questions were good. For second part of study (code review tasks), we gave the pilot participants 10 minutes for each task (30 minutes in total for all 3 tasks). They provided suggestions on how to improve the code review task to make it more realistic. For instance, they felt the time was too tight to review three contracts, and suggested having each participant review only one contract, which should be more comprehensive and realistic by having common functionalities. They mentioned that code review takes an effort, and might 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.

2 SUPPLEMENTARY FOR SECTION 4

2.1 Figure to Supplement Section 4.3: Factors considered in Development and Tools/Programming Languages

We include figures representing different opensource library use in Figure 1, awareness of security vulnerabilities 2 and desired tooling in Figure 3 by survey respondents.

2.2 Figure to Supplement 4.5: Task Flowcharts of Participants

This section includes flowcharts of code review process which were most common among participants. The Figure 4 and Figure 5 complement section 4.5 in main paper. Mainly the themes on *"Different approaches to code review"*. .

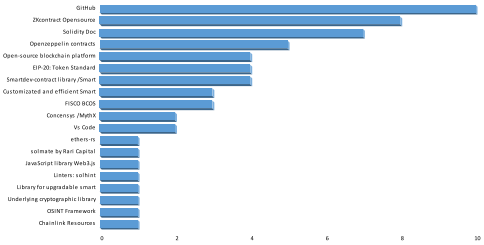

Fig. 1. Survey respondents' use of Open Source libraries/ code/standards

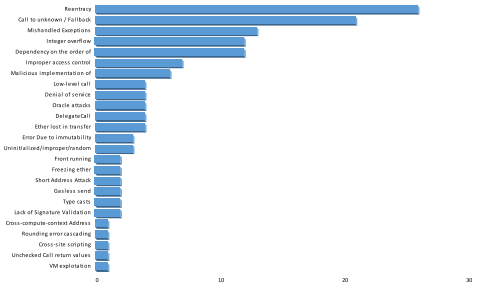

Fig. 2. Survey respondents' awareness of security vulnerabilities

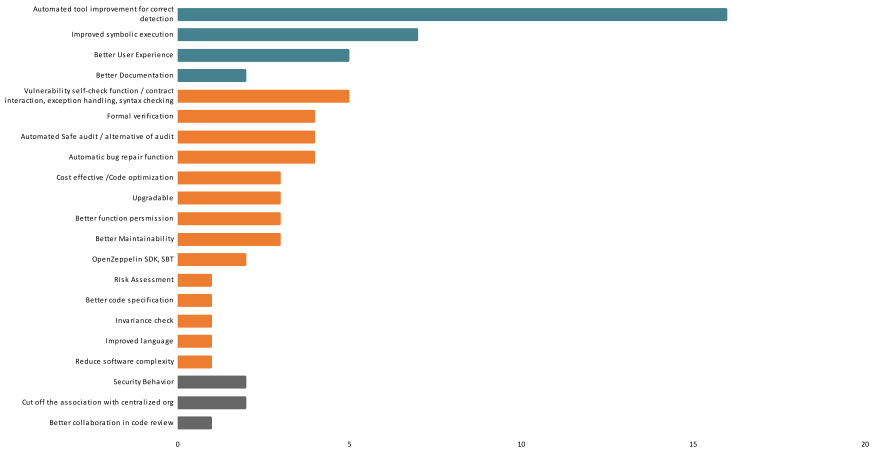

Fig. 3. Survey respondents' Desired tooling in three different categories

We included a few more examples of how other participants performed the code review tasks in this supplementary document.

2.3 Supplement Section 4.5: Code Review Improvement Suggestions: Task 1 and 2

This section includes examples of modified code snippets by participants from code review session as well as from survey respondents (Figure 12 and Figure 13) who were able to identify security vulnerabilities. We also includes code snippet from participants who identified security vulnerabilities in smart contract, however, false positive in Figure 15.

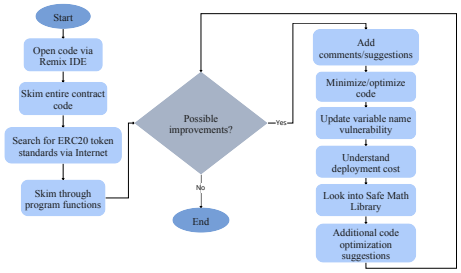

Fig. 4. Code Review Task Flowchart for P22 who manually reviewed smart contract for Task 2 and failed to identify any vulnerabilities

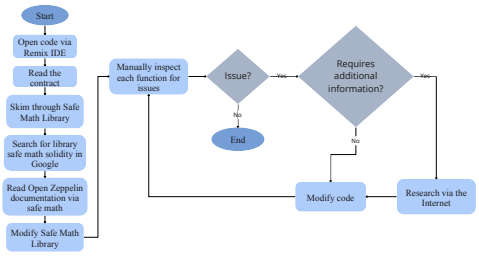

Fig. 5. Code Review Task Flowchart for P6 who manually reviewed the code and successful in identifying both vulnerabilities in Task 2

Table 1. Summary of participants' task performance in the code review task where success rate (i.e., Rate) denotes the percentage of participants who correctly identified a particular vulnerability; ✗: the participant did not identify the vulnerability; ✓: the participant identified the vulnerability. The top section of the table shows the junior (0-1 year of experience) Solidity developer participants, whereas the middle part are mid-level developers (more than 1 years -3 years) and bottom section of the table shows the senior(more than 3 years of experience) Solidity developer participants. Every participant either did Task 1 or Task 2. Task 1 includes two vulnerabilities: reentrancy and unchecked low-level call. Task 2 includes two vulnerabilities: integer overflow and improper access control. **Reen** denotes Reentrancy; **LL**: Low level call; **OF**: Integer Overflow; **AC**: Access Control vulnerabilities. We also show the review method each participant used during the task. Most participants did manual inspection of the code using an IDE or a text editor such as Remix, VS Code, Sublime, IntelliJ, or simply viewing the code on GitHub. Remix and Hardhat are development/testing frameworks for smart contracts. Three participants (P3, P15, P20) used Remix with a security plugin (sp), which does Solidity static analysis. Slither is a static analysis tool and Oyente is an analysis tool based on symbolic execution. Average time for identifying each vulnerability by developers. a_T: Average Time

ID	*Years*	*Reen*	*Time*	*LL*	*Time*	*OF*	*Time*	*AC*	*Time*	*Tool Used*
mygray		**Task 1**				**Task 2**				
mygray Success Rate		53.3%		40%		28.6%		42.9%		
mygray Average time (minutes)		11.57		9.8		8.6		12.98		
Junior (n: 10)		25%		0%		16.7%		16.7%		
P1	<1					✗		✗		Manual
P4	<1	✗		✗						Manual
P9	<1					✗		✗		Manual
P16	<1					✗		✗		Manual
P3	<1					✗		✗		Remix(sp)
P5	<1	✗		✗						Manual
P7	<1	✗		✗						Manual
P24	<1					✗		✗		Manual
P20	<1	✓	18.18	✗						Remix(sp)
P28	<1					✓	7.07	✓	8.31	Manual
mygray			a_T: 18.18				a_T: 7.07		a_T: 8.31	
Mid-Level (n: 10)		50%		33.33%		25%		75%		
P2	2	✗		✗						Manual
P11	2	✗		✗						Manual
P15	2					✗		✓	11.45	Remix(sp)
P17	2					✓	9.13	✗		Manual
P23	2	✗		✗						Manual
P18	2.5	✓	13.07	✗						Manual
P21	2.5					✗		✓	14.54	Remix+Oyente
P14	3	✓	12.44	✓	10.12					Hardhat+Slither
P25	3	✓	6.23	✓	10.53					Manual+Remix (sp)
P26	3					✗		✓	15.47	Manual
mygray			a_T: 10.58		a_T: 10.53		a_T: 9.13		a_T: 13.82	
Senior (n: 9)		80%		80%		50%		50%		
P19	3+	✓	13.5	✗						Manual
P22	3.5					✗		✗		Manual
P13	4					✗		✗		Manual
P29	4	✗		✓	10.37					Manual+Remix (sp)
P8	4					✓	9.17	✓	14.01	Manual
P10	4+	✓	5.08	✓	8.08					Manual
P27	4+	✓	12.13	✓	11.56					Manual+MythX
P12	4.5	✓	11.98	✓	8.16					Manual
P6	5+					✓	9.08	✓	14.15	Manual
mygray			a_T: 10.67		a_T: 9.54		a_T: 9.13		a_T: 14.08	

Table 2. Survey Respondents' Task Success Rate and Task Time. Average time for identifying each vulnerability by developer a_T: Average Time; *SR* = Success Rate

ID	*Years*	*Reentancy*	*Overflow*	*Access Control*	*Frontrun*	*Flashloan*	*Tool Used*
mygray Overall SR		**18.9%**	**26%**	**15.63%**	**17.65%**	**24.32%**	
mygray Overall a_T		a_T: 7.95	a_T: 9.29	a_T: 10.27	a_T: 10.01	a_T: 12.2	
P1	1					35.51	Remix
mygray Junior						a_T: 35.51	
mygray SR		**0%**	**0%**	**0%**	**0%**	**33%**	
P2	2	9.87					Remix
P3	2			4.92			Remix
P4	2			8.59			Vs code
P5	3	6.04					Notepad
P6	3	8.03					Remix
P7	3			11.94			Truffle Suite
P8	3				14.6		Remix
P9	3					8.78	Remix
mygray Mid-Level		a_T: 7.98		a_T: 8.48	a_T: 14.6	a_T: 8.78	
mygray SR		**27%**	**0%**	**23%**	**12.5%**	**16.67%**	
P10	4		7.14				Sublime Text
P11	4		11.83				Vs Code
P12	4			7.61			Dreaweaver
P13	4				7.59		Remix
P14	4					6.61	Remix IDE, Truffle
P15	5	7.86					Vs Code
P16	5		8.19				Vs Code
P17	5		6.57				None/Manually
P18	5		13				None/Manually
P19	5		6.49				Vs Code
P20	5					7.14	Remix IDE
P21	6		11.64				Remix IDE
P22	6		9.49				Sublime Text
P23	6					10.21	Slither+MythX
P24	6					7.03	Remix IDE
P25	6					9.3	None/Manually
P26	7	7.005					None/Manually
P27	7				7.3		Vs Code
P28	7					15.52	None/Manually
P29	8	10.42					None/Manually
P30	9				9.92		None/Manually
P31	9				7.07		None/Manually
P32	9				13.62		None/Manually
P33	10					9.45	None/Manually
P34	10	6.4					Oyente
P35	10				18.3		Remix IDE
mygray Senior		a_T: 7.92	a_T: 9.29	a_T: 12.95	a_T: 9.1	a_T: 9.32	
mygray SR		**18.18%**	**35%**	**12%**	**21%**	**25%**	

```
modifier onlyOwner() {
  require(msg.sender == address(owner)) ;
  _;
}
```

Fig. 6. Security vulnerability identification (access control) and modification by P6 during the task2. Tools used: Remix, manual inspection.

```
    function withdraw(uint256 amount) public
    ↪  returns (bool success) {
        // typically a lot of contracts revert
        ↪  in this case
        // revert('Insufficient balance');
        ↪  instead of `return false`
        if (balanceOf[msg.sender] < amount)
        ↪  return false;
        balanceOf[msg.sender] -= amount; //
        ↪  safe because check above
        totalSupply -= amount;
       (bool success,) = msg.sender.call{value:
        ↪  amount}(amount);
        require(success);
        return true;}
```

Fig. 7. Security vulnerability identification (re-entrancy) and modification by P10 during the task. Tool used: Remix.

```
    function withdraw(uint256 amount) public
    ↪  returns (bool success) {
        if (balances[msg.sender] < amount)
        ↪  return false;
        balances[msg.sender] -= amount;
        _totalSupply -= amount;
        //There is some danger in using send:
        ↪  check
        //https://docs.soliditylang.org/
        if (!msg.sender.send(amount)) {
        //vulnerable to reentrancy attack:
        //https://docs.soliditylang.org/
            balances[msg.sender] += amount;
            _totalSupply += amount;
            return false;}
        return true;}  }
```

Fig. 9. Security vulnerability identification (re-entranccy) and modification by P12 during the task. Tool used: Remix.

```
    function transferFrom(address from,
    ↪  address to, uint256 tokens) override
    ↪  public returns (bool success) {if
    ↪  (balanceOf[from] >= tokens &&
    ↪  allowance[from][msg.sender] >=
    ↪  tokens) {
           // no use of safemath
           // todo: is this safe, with the
           ↪  caveat that total supply of
           ↪  15e23 + all ether <
           ↪  type(uint256).max
           balanceOf[to] += tokens;
           balanceOf[from] -= tokens;
           allowance[from][msg.sender] -=
           ↪  tokens;
           emit Transfer(from, to, tokens);
           return true;} else {
           // typically a lot of contracts
           ↪  revert in this case
          // revert('Insufficient allowance');
           ↪  instead of `return false`
           return false; }}
```

Fig. 8. Security vulnerability identification (low-level call) and modification by P10 during the task. Tools used: VS Code, manual inspection.

```
    function transferFrom(address from, address
    ↪  to, uint256 tokens) override public
    ↪  returns (bool success) {
        //modification: adding require here
        require(from!=address(0), "Empty
        ↪  address is not allowed");
        require(to!=address(0), "Empty address
        ↪  is not allowed");
        if (balances[from] >= tokens &&
        ↪  allowed[from][msg.sender] >=
        ↪  tokens && tokens> 0) {
            balances[to] += tokens;
            balances[from] -= tokens;
           allowed[from][msg.sender] -= tokens;
            emit Transfer(from, to, tokens);
            return true;
        } else { return false; }}
```

Fig. 10. Security vulnerability identification (low-level unchecked calls) and modification by P12 during the task. Tools used: VS Code, manual inspection.

```
 //Might be a vulnerability- something to do with miners reordering approve txs
 //I kinda remember approve funtions having a "set to 0" kinda logi
//look deeper later
        function approve(address spender, uint256 tokens) override public returns (bool success) {
        allowed[msg.sender][spender] = tokens;
        emit Approval(msg.sender, spender, tokens);
        return true;
    }
```

Fig. 11. False positive Security vulnerability identification for task1 by P1

```
1. controls implementation so withdraw ()  can
↪  only be triggered by authorized parties
2. replace function construct() with
↪  constructor()
3. use error ErrorName(params) with revert
↪  ErrorName(params)
4. _to.transfer(msg.value) becomes
↪  _to.call{value: msg.value}("")
```

Fig. 12. Security vulnerability identification (Unprotected Ether Withdrawal relating to improper Access control) and modification suggestions provided by survey respondent Tool used: Remix.

```
    ERC20 approve() front running
commit reveal hash scheme
Add a field to the inputs of approve
Set approval to zero before changing them.
For permission validation, either require(this
↪  == msg.sender) or require(owner ==
↪  msg.sender)
```

Fig. 13. Security vulnerability identification (Front-Running) and modification by P12 during the task. Tools used: VS Code, manual inspection.

```
 //Might be a vulnerability- something to do with miners reordering approve txs
 //I kinda remember approve funtions having a "set to 0" kinda logi
//look deeper later
        function approve(address spender, uint256 tokens) override public returns (bool success) {
        allowed[msg.sender][spender] = tokens;
        emit Approval(msg.sender, spender, tokens);
        return true;
    }
```

Fig. 15. False positive Security vulnerability identification for task1 by P1

```
 function batchTransfer(address[] _receivers, uint256 _value) public whenNotPaused returns (bool) {
    uint cnt = _receivers.length;
    require(cnt > 0 && cnt <= 20);
    uint256 amount = mul(uint256(cnt), _value);

    // below is possible taken care of by super.transfer
    require(_value > 0 && balances[msg.sender] >= amount);

    // balances[msg.sender] = balances[msg.sender].sub(amount);
    for (uint i = 0; i < cnt; i++) {
        // checks if receivers are address(0x0) --> revert if true
        super.transfer( _receivers[i], _value);
        // balances[_receivers[i]] = balances[_receivers[i]].add(_value);
        // emit Transfer(msg.sender, _receivers[i], _value);}
    return true;
    }
}
```

Fig. 14. Part of the contract code P6 reviewed and modified. It had an overflow vulnerability where local variable is calculated as the product of cnt and $\_value$. By having two $\_receivers$ passed into $batchTransfer$, with that extremely large $\_value$, attacker can overflow amount and make it zero.

3 INTERVIEW PROTOCOL

3.1 Recruitment Scripts-Interviews

We are a group of researchers at X and we are recruiting people who have experience with solidity smart contracts for a study that aims to better understand developers' experience in developing smart contracts. The interview will take about an hour in which the researchers will ask questions about their security practices as well as there will be a code review task. There is a 30$ compensation for each participant. If you are interested, feel free to email at xxx@xxx.com.

Fill out the short screening survey to help us identify if you are eligible to participate in this study.

3.2 Programming Background

(1) What programming languages do you use regularly?
(2) What kinds of applications or software have you worked on? (Front-end/back-end/ android app/ios app?) Follow-up: What about more recent apps you worked on?

3.3 (Smart Contract Developing) - Solidity

This section of questions is more on their personal experience and stories of handling security issues on smart contract writing

(1) How would you explain smart contracts to a lay person?
(2) What role do you typically play in crypto currency projects? (Do you write smart contracts/front end?)
(3) How did you first get to know about smart contracts? How did you learn smart contract development? (Can you give me an example?)
(4) What kind of tools are you using for smart contract development? (Follow-up: IDE, Plugins, API...)
(5) How do you usually develop smart contracts? Can you walk me through the recent smart contract you developed? (Can you give me some examples?)
    (a) Did you develop it by yourself or with a team?
    (b) What are the steps you went through during development?
    (c) What primary factors did you consider during smart contract development? How do you prioritize those factors?
    (d) Do you use/follow any code writing standards? If so, what are they?
(6) Do you have any experience in deploying smart contracts? Where did you deploy it? Was it a production system or just for fun/trying things out?
(7) Do you consider security when developing smart contracts? If so, how? What types of security related practices are you aware of? Do you follow any of those practices? Could you give me a concrete example?
(8) What are the most frequent security issues you have encountered? Please tell us about any incident that you can remember when your developed smart contract was exposed to certain security vulnerabilities? What were the security vulnerabilities? How did you/ your team handle this issue?
    Please tell us how you did feel about that security incident. (nudging: Severity level? Company budget? Other team members' feelings?) What would you do differently now, to avoid such an incident?
(9) Have you encountered any challenges in ensuring the security of your smart contracts? (if yes, What kinds of difficulties or challenges do you face in ensuring security of smart contracts?/ if no, Do you generally anticipate any challenges) How do you currently overcome these challenges? Can you give me a concrete example?

This section of questions is on their current practice (standards, tools, policy)

(10) Do you/your team consider security during smart contract development? If so, how?
(11) What types of methods/tools are you/your organization currently using to ensure the security of smart contracts?
    (a) Have you used any automated security analysis tools? Can you explain more about the tools? (Is it easy or hard to use? How does it balance security and usability? What do you think makes tools or APIs more usable, e.g., better documentation? What kinds of tools do you want to help you ensure security of smart contracts?)
(12) Do you perform code review / testing on your (or someone else's) smart contract code base?
    (a) Can you share your experience on performing code reviews/testing for smart contracts?
(13) What are the information sources you turn to for help with regard to security issues?

3.4 Developer Tasks (25min)

We have the next activity which is code review. For this activity, can you share your screen and allow us to record the screen to better understand your experience. For this activity, your task is to 1) review the code and then 2) identify the areas for improvement and modify the code, 2) Explain the improvements I am sharing the Link in the chat where you can find your assigned tasks. For this task, you have 25 minutes. For this review activity, you can use any tools you want.

3.5 Code Review Tasks.

GitHub Repo: https://github.com/AccountProject/Developer_Study_SC

3.6 Wrap up

I really appreciate all the time you've given me. As we wrap up, let me summarize some of the key points I've learned about your experience here.

(1) Can you share your experiences in doing the code review tasks?
(2) If we are to develop tools for ensuring security of smart contracts, what kind of features would you like to have?
(3) Is there anything else regarding security practices in smart contract development that you would like to add? And Does this study have changed your perspectives and/or attitudes towards current security practices?
(4) Can you provide any feedback on this study?
(5) Can you suggest another interested person identifying as a smart contract developer who would like to get involved with the study?

Thank you for your time. If you have any questions or concerns about this research, feel free to contact us!

4 SURVEY PROTOCOL

This survey will take about 30 minutes. To make it easier for you to complete, you will have 24 hours to finish it. Thanks!

4.1 Section1: Smart Contract Security Practices

(1) Are you 18 years or older? [yes, no]
(2) Do you agree that your survey response to be used in presentations for academic or research purposes? [yes, no]
(3) Do you identify yourself as smart contract (solidity) developer or identify as having familiarity with smart contract (solidity) development? [yes, no, other]

(4) What is your gender? [Male, Female, Other]
(5) What is your age range? [18-25, 25-34, 35-44, 45-54, 55-64, 65-74, 75+, Others]
(6) What is your highest level of educational achievement? [Less than high school, Some high school, High school graduate, Diploma or the equivalent (for example: GED), Some college credit, Trade/technical/vocational training, Associate degree, Bachelor's degree, Master's degree, Professional degree, Doctorate degree, No degree, Others]
(7) Which category best describes you? [White (Eg: German, Irish, English, Italian, Polish, French, etc), Hispanic, Latino or Spanish origin (Eg: Mexican or Mexican American, Puerto Rican, Cuban, Salvadoran, Dominican, Colombian, etc), Black or African American (Eg: African American, Jamaican, Haitian, Nigerian, Ethiopian, Somalian, etc), Asian (Eg: Chinese, Filipino, Asian Indian, Vietnamese, Korean, Japanese, etc), American Indian or Alaska Native(Eg: Navajo nation, Blackfeet tribe, Mayan, Aztec, Native Village or Barrow Inupiat Traditional Government, Nome Eskimo Community, etc), Middle Eastern or North African (Eg: Lebanese, Iranian, Egyptian, Syrian, Moroccan, Algerian, etc), Native Hawaiian or Other Pacific Islander (Eg: Native Hawaiian, Samoan, Chamorro, Tongan, Fijian, etc), Some other race, ethnicity or origin, Prefer not to say]
(8) Are you working full-time in the cryptocurrency or blockchain industry? [Yes, No, Other]
(9) How many years of smart contract development experience do you have? [Number entry]
(10) Which programming language(s) do you use for smart contract development? [Select all that apply] [Solidity, Viper, Rust, Other, please specify]
(11) What's your main role in blockchain application development? (please select all that apply) [Select all that apply] [Protocol development, Smart contract development, Research, Community engagement, Dapps UI/UX, Security assessment, Tokenomics, Others, please specify]
(12) What are the main information sources that you use to learn about smart contract (solidity) development? (please select all that apply) [Google search, Solidity Documentation, Youtube, OpenZepplin, Stack Overflow, ConsenSys, GitHub, Other, please specify]
(13) What are the main tools that you use in (solidity) smart contract development? (please select all that apply) [Ganache, Remix IDE, Geth, HardHat, Waffle, Truffle, Vscode, Other, please specify]
(14) What are the Open Source libraries/code/standards, if any, that you use during smart contract development? [open text]
(15) Do you have experience with reviewing (solidity) smart contract code? [Yes, no, Other, please specify]
(16) Do you have code review experience with other programming languages? [Yes, please specify, No, Other, please specify]
(17) In your opinion, what is important in smart contract development? [1: less important; 5: more important] [Functionality correctness, Gas efficiency, Security, Code optimization, Maintainability, User experience]
(18) Where have you learned about smart contract security? (select all that apply) [Solidity documentation, News, Tutorials, Workshops, Hackathon, Other, please specify]
(19) What tools or resources do you use for smart contract security? (select all that apply) [Manual inspection, Slither, MythX, Remix, HardHat, Security Plugins, please specify, Other, please Specify]
(20) Are you aware of any security vulnerabilities in the context of smart contracts? Please specify.
(21) Imagine your dream tool for smart contract security, what features should this tool have?

4.2 Section 2: Code Review Block

The smart contract codes can also be found in **Github repo** https://github.com/AccountProject/Developer_Study_SC

(1) Do you think this code needs further improvement? [yes, no, I don't know] [if "no", skip to Q4, if "I don't know", skip to Q4]
(2) If yes, what issue(s) does this code have? [Please explain briefly the areas of improvement]
(3) How do you solve the issue(s)? [Please write down the improved code or describe your improvement strategy in the text box]
(4) What resources or tools did you use for this task?
(5) How would you rate the difficulty of this task? (1 - not difficult at all, 10 - most difficult)

4.3 Interview Coding

This section includes the coded themes of interview responses and categorizes those into main theme categories. Figure 16 presents categories and sub-categories of different interview codings.

Perceptions of smart contract security.

- **Smart contract is secure program**: This means smart contract is already secure, and doesn't need security consideration.
- **Security is secondary / Contract security often not a top priority**: For this coding, participants didn't consider security as the primary factor.
- **Smart Contract is integrity of entity and rule of transaction**: Participants for this coding mentioned about the blockchain-based resilient system that allow for information integrity of rules and agreement.
- **Smart Contract is a way of being innovative/interactive in blockchain**: Participants in this theme considered smart contract as a revolution and an open source infrastructure where they can interact with different smart contracts, dapps, etc
- **Smart contract languages/ framework/ API is already secure**: For this theme, participants mentioned that language and supporting itself secure or showed optimism towards existing framwork.
- **Security means Functional correctness:** For this, participants perceive functional correctness including, logic, function, and methods as security.
- **Security often handled by others or auditing**: For this theme, participants showed overreliance towards audit and thought it was not to be considered during the development stage.
- **Adversarial mindset in security consideration:** This theme indicates security perception to have an adversarial mindset towards security vulnerabilities while writing code, including, consideration of scenarios that can lead to security vulnerabilities.
- **Security is the main objective**: Participants who mentioned security as their main priority.
- **Security means impact (how big/small the project is)**: For this, participants mentioned that security depends on the size of the project, If the project is not significant, then security is not important.
- **Security is not important when projects are deployed on testnets:** Participants in this theme didn't consider security when the project is deployed on testnet since it is not exposed to vulnerabilities.

Security Challenges.

- Deployment Challenges due to Gas cost for smart contract
- Deployment Challenges due to hash
- Deployment Challenges related to code optimization
- Deployment challenges in mainnet

- Limitation in tools for Smart contract security detection (besides the known ones)
- Challenges for new developer to start from scratch for new developers
- Solidity language limitations makes contract security hard

Practice by Developers

- Audited/vetted code re-use encouraged for cost effectiveness
- Deployment experience (in testnet and mainnet)
- Individual /team security practice in SM code use, code auditing, research and writing
- Speed (getting product in market) as important factor in development
- Resources for learning smart contract coding (solidity doc, youtube, google, concensys, etc.)
- Resources for learning security vulnerabilities (report published by different org, twitter, news, etc)

Security strategies.

- **Some conformity to standard for code writing:** Participants mentioned using existing standards, such as openzepplin, concensys, etc
- **Documentation (general SE best practices)**: Participants mentioned the document and general software practice for ensuring security.
- **Common and edge case consideration in SM security**: Participants mentioned being well informed of known security vulnerabilities to ensure security
- **Manually inspection is the best way to code review**: Participants mentioned manual inspection as their main way to code review and ensure security.
- **Input Validation: Extra code size function:** This is a form of security testing.
- **Input validation (use logic/reasoning)**: This is a form of security testing, leaning towards functional correctness.
- **Input validation Modular structure**: This is a form of security testing.
- **Iterative process: fuzzing, testing:** This is a form of security testing.
- **Truffle tests**: Participants mentioned using the truffle framework for ensuring security.
- **Invariants, fuzz testing, unit test, functional test, and integration testing:** This is a form of security testing.
- Keep abreast w/ smart contract community and news (e.g., eth foundation, openzepplin, paradigm)
- **Use existing (vetted) libraries**: participants mentioned Openzepplin for ERC20 contract
- **Use more secure language, e.g., Rust**: Participants consider RUST to be more secure than solidity
- **Write/run one's own simulations**: This theme presents participants' response of building own simulation for security.
- **Create one's own error codes (byte code dictionary)**
- Constant refactoring and improving code
- **write the most simple code** that you can
- Making the contract easily **upgradable**
- Ensure good **access control**
- **Draw state machine**: To make sure of the workflow of the code and ensure security.
- **Automatic linter (static analysis)**: This present participants using linter plugin to ensure indentation.

Security Concerns.

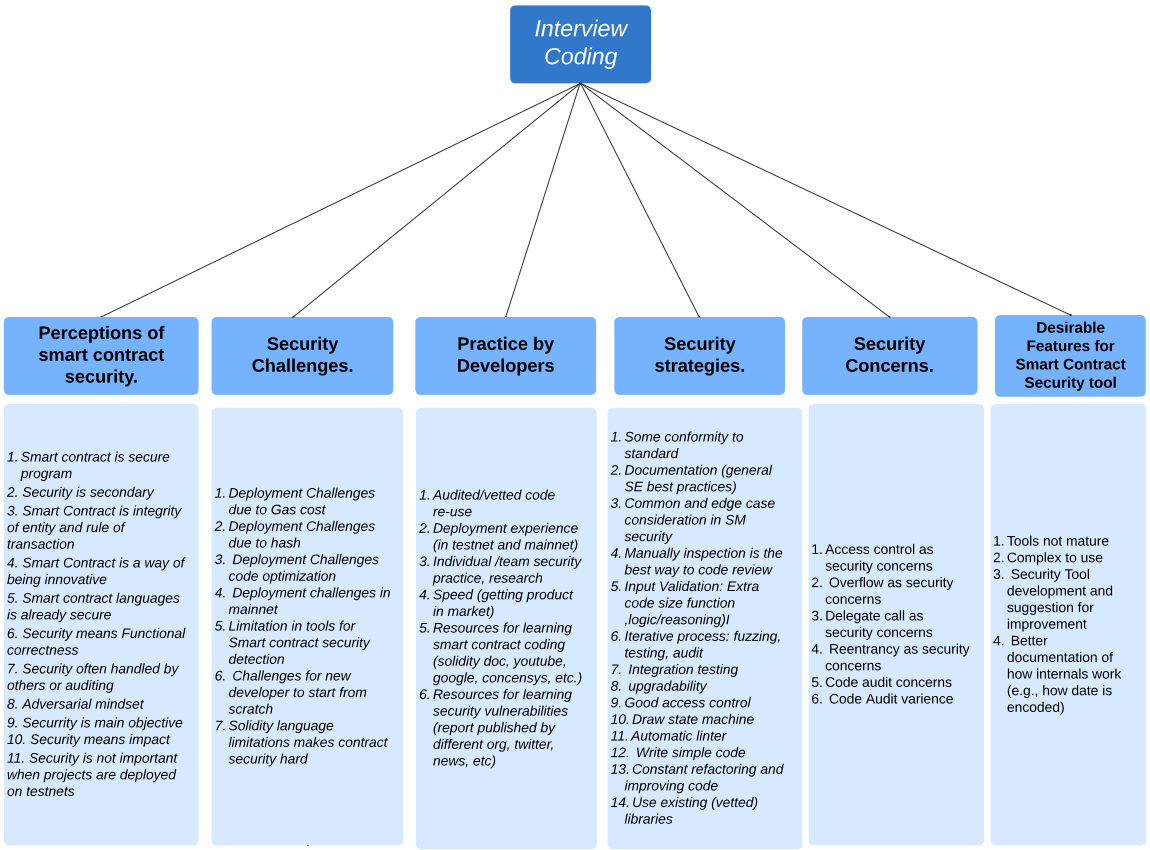

Fig. 16. Interview Coding

- Access control as security concerns
- overflow as security concerns
- delegate call as security concerns
- reentrancy as security concerns
- code audit concerns
- Code Audit varience

Desirable Features for Smart Contract Security tool.

- tools not mature
- complex to use
- security Tool development and suggestion for improvement
- Better documentation of how internals work (e.g., how date is encoded)