When Contracts Meets Crypto: Exploring Developers' Struggles with Ethereum Cryptographic APIs

Jiashuo Zhang
School of Computer Science
Peking University
Beijing, China
zhangjiashuo@pku.edu.cn

Jiachi Chen*
Sun Yat-sen University
Zhuhai, China
chenjch86@mail.sysu.edu.cn

Zhiyuan Wan
Zhejiang University
Hangzhou, China
wanzhiyuan@zju.edu.cn

Ting Chen
University of Electronic Science and Technology of China
Chengdu, China
brokendragon@uestc.edu.cn

Jianbo Gao
School of Computer Science
Peking University
Beijing, China
gaojianbo@pku.edu.cn

Zhong Chen
School of Computer Science
Peking University
Beijing, China
zhongchen@pku.edu.cn

ABSTRACT

To empower smart contracts with the promising capabilities of cryptography, Ethereum officially introduced a set of cryptographic APIs that facilitate basic cryptographic operations within smart contracts, such as elliptic curve operations. However, since developers are not necessarily cryptography experts, requiring them to directly interact with these basic APIs has caused real-world security issues and potential usability challenges. To guide future research and solutions to these challenges, we conduct the first empirical study on Ethereum cryptographic practices. Through the analysis of 91,484,856 Ethereum transactions, 500 crypto-related contracts, and 483 StackExchange posts, we provide the first in-depth look at cryptographic tasks developers need to accomplish and identify five categories of obstacles they encounter. Furthermore, we conduct an online survey with 78 smart contract practitioners to explore their perspectives on these obstacles and elicit the underlying reasons. We find that more than half of practitioners face more challenges in cryptographic tasks compared to general business logic in smart contracts. Their feedback highlights the gap between low-level cryptographic APIs and high-level tasks they need to accomplish, emphasizing the need for improved cryptographic APIs, task-based templates, and effective assistance tools. Based on these findings, we provide practical implications for further improvements and outline future research directions.

KEYWORDS

Ethereum, Smart Contracts, Empirical Study, Cryptography, API Usability

*corresponding author

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639131

ACM Reference Format:
Jiashuo Zhang, Jiachi Chen, Zhiyuan Wan, Ting Chen, Jianbo Gao, and Zhong Chen. 2024. When Contracts Meets Crypto: Exploring Developers' Struggles with Ethereum Cryptographic APIs. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639131

1 INTRODUCTION

Cryptographic techniques, leveraging their capabilities to ensure the security of data, computation, and communication, have brought a wide range of innovations to smart contracts. By employing advanced cryptographic tools, smart contracts have effectively promoted on-chain identity authentication [12, 60], private computation [10, 88, 89], and numerous other promising functionalities [23, 96, 104]. These capabilities significantly enhance the flexibility and operational scope of on-chain applications.

To enable on-chain cryptographic practices, Ethereum officially introduces a set of system-level cryptographic APIs and allows developers to implement various cryptographic tasks based on them. For example, the verification of ECDSA signatures [52] can be implemented by composing the `KECCAK256` and `ECRECOVER` APIs [101] in sequence. These crypto APIs effectively reduce the gas cost associated with complex cryptographic operations and attract emerging applications from the industry.

However, while the advent of Ethereum crypto APIs theoretically facilitates on-chain cryptographic practices, the practical challenges related to their utilization and integration may hinder developers from accomplishing their cryptographic tasks. Since smart contract developers may not necessarily be cryptographic experts, expecting them to directly interact with fundamental cryptographic operations such as elliptic curve addition could pose usability challenges and potential misuse risks. For example, a security team reported that 52 smart contracts suffer signature replay attacks due to their vulnerable signature verification implementations [5]. To prevent such difficulties from constraining the broader on-chain application of cryptographic tools, it is essential to understand the obstacles developers encounter and provide better support for them.

While usability of traditional crypto libraries has been well-studied [61, 63, 72], the real-world usage of Ethereum crypto APIs

remains unexplored. This paper aims to characterize obstacles in using Ethereum crypto APIs, elicit underlying reasons behind them, and derive potential solutions to mitigate them. Specifically, we focus on the following research questions.

RQ1. What are the common cryptographic practices in smart contracts?

Understanding the tasks developers need to perform is the first step to understand the obstacles they face [63, 80]. To characterize cryptographic practices in smart contracts, we conducted a study on 91,484,856 Ethereum transactions and 500 crypto-related smart contracts. Our findings reveal the prevalence, classification, and distribution of common cryptographic tasks in Ethereum smart contracts, including signature [65], commitment [24], message digest [75], random number generator [99], proof of work [100], and zero-knowledge proof [91]. Such information can guide solution designs and align community support with real-world demands.

RQ2. What obstacles, if any, do developers face when accomplishing on-chain cryptographic tasks?

Existing research highlighted that developers struggle with cryptographic tasks in traditional applications [42, 61, 63], but the challenges specific to smart contract cryptographic practices remain unknown. Through the analysis of 483 StackExchange posts, we confirmed that smart contract developers face obstacles in cryptographic practices. We further classified these obstacles into five categories, i.e., *Knowledge, Roadmap Identification, Template Usage, API Usage*, and *Security*. These obstacles underline the challenges in understanding, implementing, and securing on-chain cryptographic practices, and highlight the need to provide better support for developers.

RQ3. How do real-world practitioners perceive these obstacles and what improvements do they desire?

While our analysis of RQ2 confirmed the presence of challenges in smart contract cryptographic practices, the root causes of these obstacles and possible solutions to mitigate them remain unexplored. To understand obstacles and solutions from the practitioners' perspective, we conducted an online survey involving 78 smart contract practitioners. The results suggest that the current crypto APIs might be too low-level for developers. 57.8% of participants face obstacles in identifying detailed implementation steps for specific cryptographic tasks, *e.g.*, deciding which crypto API should be used, before they actually operate the underlying APIs. The practitioners' feedback highlights the need for improved crypto APIs, task-based templates, effective testing/audit tools, and easy-to-understand documentation.

In the following, we summarize our main contributions:

- We categorize common cryptographic tasks in smart contracts and explore their prevalence and distribution. The results provide the first close look at on-chain cryptographic practices and can derive guidance for further studies.
- We reveal the obstacles faced by developers in accomplishing on-chain cryptographic tasks and investigate them from the practitioners' perspective. Our findings shed light on the underlying causes of these obstacles and provide insights for future tools and solution designs.
- We publish our datasets and results at https://zenodo.org/records/10074040 to facilitate further studies.

2 BACKGROUND

2.1 Ethereum Virtual Machine (EVM)

The Ethereum Virtual Machine (EVM) serves as the runtime environment to execute Ethereum smart contracts [101]. During the execution of a contract, EVM executes a number of specific operations known as opcodes, modifies its state (including the stack, memory, and storage), and records the execution results onto the blockchain.

Opcode. Specifically, each opcode refers to a specific operation that transitions the EVM from the current state to the next. These operations could involve writing data to the blockchain, reading the blockchain state, calling another contract, and other operations. For example, by using the `STATICCALL`/`CALL`/ `CALLCODE`/`DELEGATECALL` opcode, a smart contract can interact with other contracts and execute their codes. Using the `SLOAD`/`SSTORE` opcode, the smart contract can read the data from the EVM storage to the stack or store the data from the stack to the storage. The detailed definition of each opcode is available in the Ethereum yellow paper [101].

Precompiled Contracts. Except for these opcodes, Ethereum also introduces a set of precompiled contracts [101] to support other advanced functionalities, such as cryptographic operations in smart contracts. Different from user-defined smart contracts implemented in Solidity, precompiled contracts are *low-level extensions* of EVM, implemented in the same languages as EVM (*e.g.*, Go or C++). In this case, the computation cost of the precompiled contracts can be highly optimized, thus making complex cryptographic operations feasible in smart contracts. Any contract can use the `STATICCALL`/`CALL`/`CALLCODE`/`DELEGATECALL` opcode to call precompiled contracts and execute their functionalities.

2.2 Crypto APIs in EVM

To reduce the cost of cryptographic operations, Ethereum provides a set of system-level crypto APIs in EVM. Specifically, there are one EVM opcode (*i.e.*, `KECCAK256`) and eight precompiled contracts [101] (*i.e.*, `ECRECOVER`, `SHA256`, `RIPEMD160`, `MODEXP`, `ECADD`, `ECMUL`, `ECPAIRING`, `BLAKE2F`), providing cryptographic functionalities to smart contracts.

According to the Ethereum yellow paper [101] and Ethereum improvement proposals (EIP-152 [92], EIP-196 [78], EIP-197 [18], EIP-198 [16]), current APIs mainly provide basic operations for hash, signature, and zero-knowledge proof (ZKP) tasks. Specifically, `KECCAK256`, `SHA256`, `RIPEMD160`, `BLAKE2F` provide four common hash functions [92, 101], `ECRECOVER` and `MODEXP` enable ECDSA and RSA signature verification [16, 101], and `ECADD`, `ECMUL`, `ECPAIRING` are officially stated as "ZKP-related precompiled contracts" [18, 78].

KECCAK256, SHA256, RIPEMD160, BLAKE2F. These four APIs enable four different hash functions in smart contracts. Specifically, the `KECCAK256` opcode was introduced since Frontier (2015.07) [31], computing KECCAK256 [45] hash functions in smart contracts. The `SHA256` and `RIPEMD160` precompiled contracts were introduced since Frontier [31], implementing the standard SHA256 [73] and RIPEMD160 [25] hash functions, respectively. The `BLAKE2F` was introduced since the Istanbul hardfork (2019.11) [33],

to support interoperability between EVM and Zcash [50] and also introduce an alternative SHA3-finalist hash function [66, 92].

ECRECOVER, MODEXP. These two precompiled contracts enable efficient signature verification in smart contracts. Specifically, ECRECOVER provides the public key recovery functionality for ECDSA signatures [52] based on the SECP-256k1 elliptic curve [98]. It was introduced since Frontier [31] to enable on-chain ECDSA signature verification. MODEXP was introduced since the Byzantine hardfork (2017.10) [32]. It provides big integer modular exponentiation operations to support efficient RSA signature verification [79], and other number-theory-based cryptography [16].

ECADD, ECMUL, ECPAIRING. ECADD, ECMUL, and ECPAIRING were introduced since the Byzantine hardfork [32]. They provide addition, scalar multiplication, and pairing operations on the *alt_bn128* elliptic curve respectively to support pairing-based zero-knowledge proof systems such as Groth16 [43] and Plonk [36].

2.3 Card Sorting Approach

To analyze the contract source codes and StackExchange posts, we employed card sorting, a common approach for organizing information into logical groups [102]. Initially, users create a card for each item, including any relevant information for classification purposes. Then they assign each card to an appropriate category in a bottom-up manner. There are three types of card sorting based on whether the categories are predefined: open card sorting, closed card sorting, and hybrid card sorting. In closed card sorting, all categories are predefined, while in open card sorting, users define the categories themselves. The hybrid card sorting is a compromise between open and closed card sorting: it has predefined categories while also allowing users to create new ones.

3 ANSWER TO RQ1: CRYPTOGRAPHIC PRACTICES IN THE WILD

Exploring common practices that developers need to perform helps to understand the challenges they encounter and provide specific support [63, 80]. In this section, we conducted the first study of on-chain cryptographic practices. First, by replaying 91,484,856 Ethereum historical transactions, we investigated the real-world usage of Ethereum crypto APIs and explored the prevalence of each crypto API. Second, by manually analyzing 500 crypto-related smart contracts, we categorized common cryptographic tasks developers need to implement and evaluated their distribution.

3.1 Data Collection and Processing

Transaction Execution Data. To evaluate the prevalence of each crypto API, we collected the execution data of Ethereum transactions and analyzed crypto API calls within them. We randomly sampled 91,484,856 transactions from all 1,928,853,563 transactions on Ethereum from 1 to 17,000,000 blocks (2015.07 to 2023.04), representing the entire set with a confidence level of 95% and a confidence interval of 0.01%. We used the *debug_traceTransaction* API of Go-Ethereum [41] to replay all sampled transactions and get their execution traces, including the executed opcodes and the internal states (*i.e.*, stack, memory, and storage) of EVM. Then, we analyzed the execution traces to identify the calls to crypto APIs.

Specifically, we used different methods to identify calls to the crypto-related opcode and precompiled contracts. For the eight crypto-related precompiled contracts, we analyzed the destination address of all contract call opcodes (STATICCALL, CALL, CALLCODE, and DELEGATECALL) to determine whether they call these contracts. For example, the CALL opcode takes the top seven elements of the EVM stack as input, where the second element is the destination contract that it intends to call. For the KECCAK256 opcode, we only filtered KECCAK256 cryptographic operations. While KECCAK256 is natively used in Solidity to maintain the mapping and dynamic array data types and compute the topics for on-chain events [101], we only focused on user-initiated cryptographic operations, instead of these native operations initiated by the Solidity compiler. Specifically, we filtered out native operations based on their operation characteristics. For example, for a mapping variable *data* with type $mapping(unit \Rightarrow address)$, access to $data[k]$ will be interpreted as the access to the storage slot located at $\texttt{KECCAK256}(k \cdot p)$, where p is the storage placeholder to uniquely identify the mapping and $\cdot$ is bytes concatenation. Therefore, if the output of KECCAK256 is used as the key for storage operations (SLOAD and SSTORE), it is a native operation to access the mapping and should be excluded from further analysis. The detailed patterns to identify all native operations are available in the online supplemental material [4].

As a result, we successfully identified 12,608,469 (13.8%) crypto-related transactions out of 91,484,856 transactions. The million-level transaction volume highlights the prevalence of cryptographic practices in smart contracts and the need for in-depth studies.

Contract Source Codes. To investigate the classification and distribution of cryptographic tasks in smart contracts, we collected source codes of crypto-related contracts for further analysis. First, we analyzed all contract calls in the transaction execution traces to extract contracts that initiated crypto API calls. Then, we queried Etherscan [28] to extract publicly available source codes of these contracts. During the execution of 91,484,856 transactions, we identified 87,538 crypto-related contracts, 19,243 of which have publicly available source codes on Etherscan. We did not deduplicate these contracts to faithfully characterize real-world distributions.

3.2 Results

3.2.1 The usage of crypto APIs. We conducted a quantitative analysis of the proportion of transactions using each crypto API. Table 1 shows the proportion of transactions using each crypto API. In particular, KECCAK256 and ECRECOVER, being used in 13.1% and 4.9% transactions respectively, are the two most prevalent crypto APIs. It demonstrates that basic cryptographic tasks, such as KECCAK256 hash functions and ECDSA signatures, have already been extensively used in real-world on-chain applications.

> ***Observation 1:*** *The two most commonly used Ethereum crypto APIs are* KECCAK256 *and* ECRECOVER*, utilized by 13.03% and 4.96% of transactions, respectively.*

Compared to KECCAK256 and ECRECOVER, the usage of other crypto APIs is relatively low (used in fewer than 1% transactions), indicating that cryptographic tasks other than hash and signatures are less prevalent. For example, while ZKP applications supported

Table 1: Proportion of transactions using each crypto API

Crypto API	API Interface	Proportion (%)
KECCAK256	EVM opcode (0x20)	13.037
ECRECOVER	precompiled contract (0x1)	4.958
SHA256	precompiled contract (0x2)	0.554
RIPEMD160	precompiled contract (0x3)	0.034
MODEXP	precompiled contract (0x5)	0.034
ECADD	precompiled contract (0x6)	0.022
ECMUL	precompiled contract (0x7)	0.022
ECPAIRING	precompiled contract (0x8)	0.022
BLAKE2F	precompiled contract (0x9)	0.000

by ECADD, ECMUL, and ECPAIRING have shown promising potential to enhance the scalability and privacy of smart contracts [10, 23, 26], their on-chain applications are still in the early stages.

In particular, we found that the BLAKE2F precompiled contract has rarely been used. Since being introduced in the Istanbul hard-fork (2019.11) [34], it has only been used by four unique contracts in 52 transactions. We analyzed these four contracts to understand the underlying reasons. Surprisingly, while EIP-152 [92] stated the envisioned usage of BLAKE2F as "allowing interoperability between EVM and Zcash", and "introducing more hash primitives", these four contracts only uses the latter functionality. It suggests that BLAKE2F, as well as on-chain SPV client [64, 92] envisioned by it, might not be essential for real-world interoperability requirements. Moreover, compared to implementing BLAKE-family hash functions based on BLAKE2F, developers tend to adopt ready-to-use hash functions like KECCAK256. The failure of the envisioned use cases and usability issues might cause the infrequent usage of BLAKE2F. Notably, there is even a draft-status EIP (EIP-7266) [19] proposing its removal.

Observation 2: *Since being introduced, the* BLAKE2F *API has only been used by four smart contracts. It might suggest an information asymmetry between the API designer and real-world demands.*

3.2.2 Common Cryptographic Tasks. To analyze the classification and distribution of common cryptographic tasks, we conducted a manual analysis of the source codes of crypto-related contracts and identified the tasks they implemented. Due to the large dataset and the time-consuming analysis process, we sampled 500 out of 19,243 crypto-related contracts to make the manual analysis feasible. On average, these contracts held 118.9 ETH and were called by 24,525 external transactions.

During manual analysis, two authors used the open card sorting approach to create categories for cryptographic tasks. Specifically, they followed the detailed workflow adopted in several previous studies [20, 103]. First, they randomly examined 40% contracts, identified the tasks implemented, and grouped them into relevant categories. If refining the categories is necessary, they went back to the beginning and reassigned these contracts to the new categories. After that, they independently classified the remaining 60% contracts. The agreement of their results, measured by the kappa score [22], is 0.86, demonstrating a high agreement. When a disagreement arises, they recheck their results and discuss to reach agreement on which result is more appropriate. As a result, we identified the following common cryptographic tasks.

Table 2: Common cryptographic tasks in 500 Ethereum smart contracts using crypto APIs

Tasks	# Contracts
Digital Signature	197
Commitment	130
Message Digest	87
Random Number Generator	74
Zero-knowledge Proof	9
Proof of Work	8
Other	58

Hash-based Tasks. While Ethereum crypto APIs only provides basic hash functions, the developers create even more diverse cryptographic tasks based on them. In particular, we identified four types of hash-based cryptographic tasks, namely *commitment*, *message digest*, *random number generator*, and *proof of work*. They are implemented based on four hash crypto APIs, *i.e.*, KECCAK256, SHA256, RIPEMD160, and BLAKE2F.

- *commitment*. We identified two types of hash-based *commitment* schemes, *i.e., vector commitment* (121, 24.2%) and *single-value commitment* (9, 1.8%). The former refers to Merkle Tree (and its variants) which allows committing to a vector of values and revealing subgroup values at specific positions later. It is often used for on-chain access control: the authorizer commits to the Merkle root, the entities who know a valid leaf and the corresponding Merkle proof can do sensitive operations, such as receiving airdrop tokens. The latter refers to a heuristic approach that computes the hash of the value as the commitment and reveals the value itself later.
- *message digest*. *message digest* is the most direct usage of hash functions, mapping an arbitrary-length message to a fixed-length "digest". It has been implemented in 87 (17.4%) contracts to compute constant-length storage/query indexes for arbitrary-length variables, *e.g.*, the unique identifier of a token trading order.
- *random number generator*. 74 (14.8%) contracts use hash functions as random number generators for scenarios such as NFT, gaming and gambling. They heuristically take attributes such as the block timestamp as seeds and hash the seeds to *deterministically* generate random numbers. However, since the seed is predictable to front-running miners, the random numbers generated by such schemes might be weak and insecure [77].
- *proof of work (PoW)*. 8 (1.6%) contracts introduced hash-based PoW schemes [100]. Users need to solve hash-based puzzles to prove the expended computing effort [100] before they perform certain operations. For example, to prevent sybil attacks, token contracts might require a PoW before users mint a token.

Digital Signature. Signature verification tasks enable contracts to authenticate messages from off-chain entities and enforce access control policies for on-chain operations. We found signature verification tasks in 197 (39.4%) contracts, including ECDSA

and RSA signatures. They usually follow the hash-then-sign paradigm: first, use hash crypto APIs to compute the hash of the signed message, then verify whether the signature is valid for the hash.

Zero-knowledge Proof. Zero-knowledge proof allows smart contracts to verify the results of off-chain computations without accessing the off-chain private inputs or re-executing the entire computation. They have attracted emerging attention [26] [10] to improve the scalability and privacy of on-chain computations. We identified 9 (1.8%) contracts that implement ZKP schemes, involving `ECADD`, `ECMUL`, `ECPAIRING`, and `MODEXP` crypto APIs.

Other. In addition, we included an "Other" category to group infrequent (less than 1%) tasks we identified, such as the Chainlink VRF [54]. We also found two *Solidity-specific tasks* in 55 (11.0%) contracts, *i.e.*, using hash APIs to compute function selectors (28, 5.6%) and the addresses of contracts created by the `CREATE2` opcode [17] (27, 5.4%). Since they are *solidity-specific* rather than *common* cryptographic tasks, we also classified them into the Other category.

Observation 3: *There exists diverse cryptographic tasks in Ethereum smart contracts, including digital signature, commitment, message digest, random number generator, proof of work, and zero-knowledge proof.*

4 ANSWER TO RQ2: IDENTIFYING OBSTACLES FROM Q&A POSTS

To check whether developers encounter obstacles and identify potential obstacles they face, we conducted an analysis of the posts from two influential Q&A websites, *i.e.*, Ethereum StackExchange [87] and StackOverflow [70].

4.1 Data Collection and Processing

During data collection, we collected questions from Ethereum StackExchange and StackOverflow. For Ethereum StackExchange, we manually filtered out 15 crypto-related tags (available in our supplement materials) from a total of 525 tags and included questions with these specific tags. For StackOverflow, we focused on questions tagged "Solidity" or "smartcontracts". Overall, we collected 27,264 questions, including 19,717 Ethereum StackExchange questions and 7,547 StackOverflow questions. All these questions are collected via the StackExchange Search API [30]. As both websites are affiliated with the StackExchange site network [29], we collectively refer to the posts on these two websites as "StackExchange posts".

Then, we conducted a keyword-based filtering to extract the questions related to on-chain cryptographic practices. To preserve the potential correlation between the obstacles and the crypto tasks developers try to perform, we separately collected questions for different categories of tasks. Specifically, hash-based tasks are grouped into the same category since they involve the same APIs and similar concepts. Signatures and zero-knowledge proofs are classified as the other two categories. We used the names of tasks and APIs such as "signature" and "ecrecover" as keywords to filter out questions that contained at least one of these keywords in their title or content. Finally, we obtained 656 Hash-related questions, 415 Signature-related questions, and 44 ZKP-related questions. To provide a 95% confidence level and a 5% confidence interval, we randomly sampled 243, 200, and 40 questions for Hash, Signature, and ZKP-related questions, respectively.

Table 3: Identified obstacles in StackExchange posts

Obstacles	Hash	Signature	ZKP
Knowledge	11	13	7
Roadmap Identification	20	32	17
Template Usage	3	39	5
API Usage	51	30	2
Security	10	7	3
N/A	148	79	6
# Analyzed Posts	243	200	40

Due to the exploratory nature of our study, we did not have any pre-defined categories for obstacles. Instead, we followed the same open card sorting procedure as Section 3.2.2 to categorize these questions. Specifically, two authors first randomly chose 40% cards, identified the main obstacle the poster was likely facing, and classified each post according to the identified obstacle. They also introduced an N/A category to include cards that are not related to on-chain cryptographic practices. Then, they independently categorized the remaining cards and discussed to resolve any differences in their results. The agreement of the two authors, measured by the kappa score, is 0.83, demonstrating a high agreement [22].

During data analysis, we noticed a relatively high proportion of N/A questions for Hash and Signature categories. This is because keywords such as "hash" and "signature" have multiple meanings, including those unrelated to on-chain cryptographic practices. For example, "hash" might refer to the transaction hashes, and "signature" might refer to function signatures defining the input and output of functions. However, these concepts were irrelevant to our study. We excluded the N/A category from the subsequent analysis.

4.2 Results

We identified five categories of obstacles from StackExchange posts and listed the number of posts facing each obstacle in Table 3. The categorization of obstacles is in line with previous studies [63, 72], with unique categories introduced by the specific context of Ethereum. Specifically, the meaning of these obstacles is as follows.

- Knowledge: Lack of knowledge of cryptography or blockchain.
- Roadmap Identification: Difficulties in identifying the detailed steps to implement a specific task.
- Template Usage: Difficulties in using third-party templates.
- API Usage: Difficulties in operating Ethereum crypto APIs.
- Security: Security concerns about the implementations.

We found that a portion of questions (11.6%, 10.7%, and 20.6%, of Hash, Signature, and ZKP questions, respectively) were caused by insufficient knowledge of cryptography or blockchain. It prevents posters from identifying the appropriate cryptographic tasks they need and assessing the feasibility of implementing these tasks on-chain. For example, post 56124326 [69] asked how to decode a SHA256 hash and retrieve its pre-image in Solidity. It reflects how a lack of understanding of hash led to the selection of incorrect tasks.

Observation 4: *11.6% Hash-related questions, 10.7% Signature-related questions, and 20.6% ZKP-related questions are caused by insufficient knowledge in cryptography or blockchain.*

Despite knowing the cryptographic tasks they need, developers might still encounter obstacles when attempting to implement them. Specifically, we identified three types of obstacles related to implementation issues, namely *Roadmap Identification*, *Template Usage*, and *API Usage* in Table 3. They are the main obstacles faced by 77.9% Hash-related questions, 83.5% Signature-related questions, and 70.6% ZKP-related questions.

The *Roadmap Identification* obstacle characterizes difficulties in identifying detailed steps to implement a specific task, *e.g.*, identifying the crypto API or templates to use. For example, post 73744 [85] asked for code examples and tutorials for achieving RSA signature verification within Solidity. After reviewing this question, we found that the poster is well aware that RSA signature can be implemented on-chain, but does not know the specific implementation steps.

Even after identifying the detailed implementation steps, the crypto API complexity or unclear underlying implementation might still prevent developers from implementing them as planned. Specifically, depending on whether these obstacles arise from reusing existing code templates or directly using the crypto APIs, we classified them into two categories, *i.e.*, *Template Usage* and *API usage*. *Template Usage* obstacles include difficulties in understanding how third-party templates work and solving error messages/unexpected results when reusing them. For example, in post 112807 [86], the poster inquired about encountering unexpected results while using the OpenZepplin's ECDSA template [67]. *API usage* obstacles refer to problems in using Ethereum crypto APIs, *e.g.*, what is the meaning of API parameters, and how to call the API in Solidity. For example, in post 49261 [84] titled *"How to use ecrecover() and what it is?"*, the poster asked what the parameters v, r, and s of `ECRECOVER` mean and how to extract them from a signature.

We observed that the distribution of the three implementation-related obstacles differ across different cryptographic tasks. For hash-related questions, the main obstacle is *API usage*, accounting for 68.9% of implementation issues. However, for signature and ZKP related questions, it only accounts for 29.7% and 8.3% of implementation issues, respectively. A possible explanation is: for simple tasks like Hash, developers often have a clear understanding of the detailed implementation steps and do not need to use templates. Therefore, their main challenges lie in the direct usage of the APIs. However, for tasks such as signature and ZKP that may involve compositions of different APIs, developers need more high-level guidance, such as identifying the correct sequence of API calls and understanding how existing templates work.

Observation 5: *Over half of posters (77.9%, 83.5%, 70.6% for Hash, Signature, and ZKP, respectively) face implementation-related obstacles. These obstacles arise from identifying detailed implementation steps, reusing third-party templates, and operating crypto APIs.*

While most questions focus on how to implement cryptographic tasks, we also found several posters (10.5%, 5.8%, and 8.8%, for Hash, Signature, and ZKP, respectively) asking about the security of cryptographic practices. They face difficulties in assessing the security of their implementations/designs and ask for assistance such as code review. In particular, 11 posters raise concerns about specific security vulnerabilities, *i.e.*, *weak random number generation* (7) and *signature replay* (4). After examining these questions, we found that some posters were familiar with the concepts of these vulnerabilities, but encountered difficulties in identifying protective solutions or determining if their implementations were vulnerable. Notably, compared to implementation issues, the number of security-related questions was relatively low. It implies that while StackExchange posts provide valuable guidance for implementing cryptographic tasks, they may provide relatively less security-related guidance, such as security warnings and protective solutions, to developers.

Observation 6: *A small portion of posters (10.5%, 5.8%, and 8.8%, for Hash, Signature, and ZKP, respectively) focus on making their implementations secure. The vulnerabilities they concern include weak random number generation and signature replay.*

5 ANSWER TO RQ3: THE PRACTITIONERS' PERSPECTIVE

The findings in Section 4 give us a preliminary understanding of developers' obstacles in on-chain cryptographic tasks. However, they may face two potential limitations. The first limitation stems from the fact that StackExchange questions may not fully represent the real-world obstacles encountered by developers. For example, the relatively few security-related questions on StackExchange do not definitively suggest that real-world practitioners seldom face security challenges. The second limitation pertains to the practitioners' perspective of the obstacles. While we have categorized potential obstacles, the underlying reasons behind these obstacles and possible solutions to mitigate them remain unexplored.

To address these limitations, we conducted an online survey targeting real-world smart contract practitioners, to validate the identified obstacles, investigate the root causes, delve into practitioners' perceptions, and determine the type of support they require.

5.1 Survey Design

We followed Kitchenham and Pfleeger's instructions [56] for personal opinion surveys and designed an anonymous survey to increase response rates [93]. The following section provides a brief introduction to our survey questions. For the complete questionnaire, please refer to the online supplementary material [4].

Demographics. (Q_{1-4}) We collected the following demographic information to understand respondents' background and distribution, and filter those who might not fully understand our survey.

- Smart contract practitioner? Yes / No. (Q_1)
- Main role as a smart contract practitioner. Development / Testing / Project Management / Research / Other. (Q_2)
- Experience in years. Free-text. (Q_3)
- Current country of residence. Free-text. (Q_4)

Cryptographic Practices. (Q_{5-6}) This part aims to understand the participants' experience in on-chain cryptographic practices. We inquired about the crypto APIs (Q_5) they have used and whether they are familiar with their basic concepts (Q_6).

Obstacles in Practitioners' Perspective. (Q_{7-10}) This part explores the difficulties of accomplishing cryptographic tasks from the practitioners' perspective. We assessed the perceived difficulty in accomplishing cryptographic tasks compared to other tasks in smart contracts (Q_7) and asked the rationale behind their perceptions (Q_8). Furthermore, we inquired about the obstacles contributing to the perceived difficulty, including the five identified obstacles and any others they faced (Q_9). We also asked participants how they acquired the knowledge for cryptographic tasks (Q_{10}).

Template Usage. (Q_{11-12}) This part aims to understand the practitioners' perceptions of the usage of third-party cryptographic templates. We asked whether participants use templates to implement cryptographic tasks (Q_{11}), and whether existing templates sufficiently support their tasks (Q_{12}).

Crypto API Design. (Q_{13-14}) This part investigates the practitioners' perceptions of Ethereum crypto APIs. Specifically, we explored practitioners' perceptions of the functionality and usability of the current API design (Q_{13}). We also inquired about any cryptographic tasks they intend to accomplish but lack support from current APIs (Q_{14}).

Security. (Q_{15-17}) This part focuses on practitioners' perceptions of the security aspects of on-chain cryptographic practices. We investigated the perceived difficulty associated with securing cryptographic tasks, compared to securing other tasks in smart contracts (Q_{15}) and asked for the rationale behind their perceptions (Q_{16}). Moreover, we inquired whether participants are familiar with crypto-related vulnerabilities listed in the Smart Contract Weakness Classification (SWC) List [76], thereby exploring their understanding of crypto-specific security in smart contracts (Q_{17}).

Resources and Tools. (Q_{18-19}) To identify the types of support desired by practitioners, we explored their perceptions of existing resources/tools (Q_{18}) and asked about any other types of support they may require (Q_{19}).

To reduce arbitrary responses caused by insufficient understanding of the questions, we provided an option of "I do not understand this question" for each multiple-choice question. For other questions, we explicitly stated that participants could skip them if they did not understand them. Our survey was made available in both English and Chinese, since English is the most widely used language and Chinese has the largest number of speakers worldwide. We carefully reviewed the two versions and guaranteed their consistency.

5.1.1 Survey Validation. We conducted a pilot survey with a small number of practitioners to obtain feedback on whether the questions are clear and easy to understand. The participants included our collaborators and partners working in well-known blockchain companies. Based on the feedback, we refined some questions for enhanced clarity without adding or removing any questions. We also polished our translation to further reduce ambiguity between the two language versions of the survey.

5.1.2 Participant Recruitment. We adopted a non-probabilistic [37] strategy for participant recruitment. Specifically, we conducted a keyword-based search for crypto-related smart contract repositories on Github, extracted their contributors' emails via the Github REST API [39], and sent the survey to them. Our selected keywords encompass typical on-chain application scenarios of cryptography, including "wallet", "token", "bridge", and "oracle", as well as common primitives including "hash", "signature", and "zkp". We also sent our survey to our industry partners working in well-known blockchain-related companies such as Meta and Alibaba.

We sent our survey to a total of 778 smart contract practitioners and received 78 responses from 21 countries. The response rate is 10.02%, which is decent compared to previous studies [15, 63]. We further excluded seven responses in which the respondents did not claim themselves as smart contract practitioners. The main roles of the remaining 71 respondents are research (34, 47.9%), development (32, 43.7%), testing (2, 2.8%), project management (1, 1.4%), security audit (1, 1.4%), and technical writing (1, 1.4%). Their average years of experience are 2.42 (min: 0.5, max: 7.0, median: 2.0, sd: 1.6).

5.2 Results

Cryptographic Practices. The top three most common crypto APIs used by participants are `KECCAK256`, `SHA256`, and `ECRECOVER`, utilized by 84.5%, 67.6%, and 57.7% of them, respectively. Most of the participants demonstrated familiarity with the concepts of Hash and Signature primitives (98.6% and 93.0%, respectively), while only 50.7% of participants were familiar with ZKP. Interestingly, although 50.7% of the participants claimed familiarity with ZKP, only about 10% of participants have used ZKP-related APIs (`ECADD` (12.7%), `ECMUL` (11.3%), and `ECPAIRING` (12.7%)) in practice, suggesting a gap between conceptual knowledge and practical implementation.

Obstacles in Practitioners' Perspective. We explored the perceived challenges in accomplishing on-chain cryptographic tasks. We found that 54.8% of participants believed that accomplishing cryptographic tasks is more challenging than accomplishing other tasks such as business logic. An additional 14.5% of participants believed that the challenges are comparable in magnitude, but cryptographic practices involve additional challenges arising from different aspects. Such results indicate that the technical stack required for cryptographic tasks may differ from that of common business logic, making it challenging for regular smart contract developers to handle cryptographic tasks.

> ***Observation 7:*** *In comparison to other programming tasks in smart contracts, 69.3% of participants believe that accomplishing cryptographic tasks presents additional challenges to them.*

To understand the root causes of these perceived challenges, we asked participants about the specific obstacles they encountered. Specifically, we included the five categories of obstacles identified from StackExchange posts in RQ2 as choices, while also allowing adding new obstacles through the "Other" choice. As shown in Table 4, real-world practitioners indeed face these five categories of obstacles. Additionally, none of the participants mentioned new obstacles, indicating the completeness of our classification.

The top two common obstacles reported by 64 respondents are *Roadmap Identification* and *security*. In particular, 57.8% of participants face obstacles in identifying the roadmap to implement their tasks, even before they actually begin the programming process. Such developers require more high-level guidance, such as task-based templates and easy-to-understand tutorials. Additionally,

Table 4: Obstacles faced by participants

Obstacles	# Participants	Ratio	Rank
Roadmap Identification	37	57.8%	1
Security	36	56.3%	2
Knowledge	26	40.6%	3
Template Usage	20	31.3%	4
API Usage	14	21.9%	5

even after implementing their tasks, 56.3% of participants confront obstacles in evaluating the security of their implementation.

Observation 8: *The most frequently faced obstacles is identifying detailed steps to implement specific tasks (faced by 57.8% of participants), followed by evaluating the security of the implementations (faced by 56.3%).*

To delve deeper into practitioners' perception, we conducted a free-text question, asking participants what makes cryptographic tasks challenging. While the majority (84.6%, 33/39) of responses could be categorized into the five obstacles listed in Table 4, we found six interesting comments that attribute obstacles to the design of Ethereum crypto APIs. For example, respondents pointed out that "*Solidity lacks native cryptographic libraries*", "*Cryptographic functions provided in Solidity are quite complex to use*", "*(cryptographic tasks) often require inline assembly to interact with precompiles*", suggesting that the current crypto APIs in Solidity are too low-level for them to use. Additionally, two of the six participants also highlighted that the high gas cost in Solidity hinders their cryptographic practices.

Observation 9: *Several participants attribute obstacles they encountered to the design of Ethereum crypto APIs. They perceive current APIs as too low-level to directly interact with.*

To explore the relationship among different obstacles, we conducted a Chi-squared test [48] to examine the potential correlation among the five types of obstacles. We observed statistically significant correlations (p-value < 0.05) between two pairs of obstacles, *i.e.*, *API usage* and *Template usage* (p-value = 0.042), *Knowledge* and *Roadmap Identification* (p-value = 0.021), indicating their relevance in the practitioners' perspective. The former correlation suggests that: since templates are fixed combinations of APIs, the obstacles in using templates might stem from the obstacles in using the APIs. For example, OpenZepplin's ECDSA template [67] internally calls the `KECCAK256` and `ECRECOVER` API to implement ECDSA signature verification. Therefore, using the template might require the understanding of the `ECRECOVER` API. The latter correlation suggests that having insufficient knowledge about cryptography or blockchain can directly prevent developers from identifying the specific implementation steps for their tasks. For example, if one does not understand the hash-then-sign paradigm [62] of signatures, they may not know the correct sequence of API calls for signature verification either. All other pairs are weakly correlated, indicating that they are distinct obstacles from the practitioners' perspective.

Crypto API Design. To explore the the participants' perceptions of current Ethereum crypto APIs, we asked them to rate the APIs on a 5-point Likert scale [53] (*very bad, bad, neutrality, good, very good*). Specifically, we provided two aspects of the Ethereum crypto APIs for rating: (1) Functionality: whether the functionalities of current Ethereum crypto APIs can support all cryptographic tasks developers need? (2) Usability: whether these APIs are easy to learn and use, even for non-expert developers? 55.9% and 47.1% of participants rated the functionality and usability of current crypto APIs as *good* or *very good*, respectively. Furthermore, only 36.8% of participants rated both aspects as *good* or *very good*, demonstrating a large room for improvement of the current crypto APIs.

To investigate whether the functionality of Ethereum crypto APIs fully meets real-world demands, we inquired if there were any tasks practitioners wished to accomplish but were not supported by current APIs. Ten respondents indicated that current APIs lack the functionality they need. Six of them explicitly requested support for other elliptic curves in addition to the current alt_bn128 curve supported by `ECADD`, `ECMUL`, and `ECPAIRING`. The needed curves included BLS12-381 [6], Pasta [49], and other pairing-friendly curves [7], which provide necessary operations for several currently unavailable tasks, *e.g.*, recursive zero-knowledge proof systems. Four respondents requested underlying support for cryptographic schemes, including BLS signature [14], Schnorr signature [81], and BBS group signature [13], which also require adding additional elliptic curve operations. For example, public key aggregation of BLS signature is unsupported by current APIs [46]. Although several Ethereum Improvement Proposals (EIPs) [3, 11] proposed adding new elliptic curves to Ethereum, none of them has been adopted.

Observation 10: *Only 36.8% of participants expressed satisfaction with current Ethereum crypto APIs in terms of functionality and usability. Current APIs lack support for several cryptographic tasks, especially those requiring other elliptic curves.*

Template Usage. To explore how third-party cryptographic templates fit into real-world practices, we asked participants whether they use existing templates or implement cryptographic tasks directly based on underlying APIs. The results showed that only 4.6% of participants always choose to implement tasks directly based on Ethereum crypto APIs, while 95.4% tend to use templates for some or all of the cryptographic tasks they implement. Furthermore, we inquired if there were any cryptographic tasks that participants desired to accomplish but lacked available templates. The responses included Zero-knowledge proof schemes (5 responses), Verkle proof [57] (1 response), and cryptographic accumulator [71] (1 response). Compared to tasks such as ECDSA signature verification, these tasks are relatively less used and have fewer code examples for reference. Therefore, they might be the tasks that require templates the most.

Observation 11: *Third-party templates can assist developers in implementing cryptographic tasks. Participants desire more templates, especially for emerging tasks such as zero-knowledge proof.*

Table 5: Participants' perceptions of current resources / tools

Resources / Tools	Distribution	Average	Score≥4
Official Document		3.38	44.9%
Audited Templates		3.59	52.2%
Testing Tools		3.14	34.8%
Audit Tools		3.06	31.9%

Security. This part investigated the practitioners' perceptions of the security of cryptographic practices and explored their understanding of crypto-related vulnerabilities. We found that compared to other common tasks in smart contracts, 64.1% of participants perceive additional challenges in securing their cryptographic tasks. To attribute these challenges, we further questioned the underlying reasons. Out of the 32 participants who provided feedback, 19 (59.4%) participants mentioned that cryptographic practices require cryptographic expertise and low-level operations, which bring more security issues inherently. For example, the respondents mentioned "*Easy to get in a pitfall with bytes/bits around you*", "*Only understanding cryptography can secure implementations while cryptography is not easy to learn*". Additionally, eight (25.0%) participants suggested that it is difficult to detect vulnerabilities in cryptographic implementations. For example, respondents mentioned that "*(cryptographic tasks are) impossible to verify, almost always probabilistic, no way to test*" and "*No smart contract security analyzer supports the detection of cryptographic-related bugs*". It demonstrates the necessity to inform developers of the security implications behind the Ethereum crypto APIs and provide improved techniques to detect security vulnerabilities.

Observation 12: *64.1% of participants facing additional challenges in securing their cryptographic tasks. They attribute these challenges to the complexity of cryptographic operations and the lack of detection methods for crypto-specific vulnerabilities.*

Furthermore, we inquired about the practitioners' familiarity with crypto-related vulnerabilities. In line with previous studies [51, 74, 82, 83], we focused on vulnerabilities in the Smart Contract Weakness Classification (SWC) list [76]. Analogous to the Common Weakness Enumeration (CWE) list for vulnerabilities in traditional software, the SWC list documents security vulnerabilities in Ethereum smart contracts. Specifically, it included the following five types of crypto-related vulnerabilities:

- SWC-117: Signature Malleability
- SWC-120: Weak Sources of Randomness from Chain Attributes
- SWC-121: Missing Protection against Signature Replay Attacks
- SWC-122: Lack of Proper Signature Verification
- SWC-133: Hash Collisions With Multiple Variable Length Arguments

The results showed that 30.9% of participants know all five types of vulnerabilities, while 11.6% of participants know none of them. On average, participants know 2.9 types of vulnerabilities (median: 3.0, sd: 1.8), indicating a moderate and improvable level of knowledge regarding crypto-related vulnerabilities. Among the five types of vulnerabilities, *Weak Sources of Randomness from Chain Attributes* and *Missing Protection against Signature Replay Attacks* are most familiar to participants, both of which are known by 62.3% of participants. Such understanding is essential for employing standard protection and preventing insecure cryptographic practices.

Resources and Tools. We examined whether the community support meets the needs of practitioners. Specifically, we focused on four types of resources and tools that are commonly used by contract developers [21, 95], *i.e.*, documentation, templates, testing tools, and audit tools, and asked participants to rate them on a 5-point Likert scale (*very bad, bad, neutrality, good, very good*). As shown in Table 5, the average scores for all resources/tools were below 4 (*good*). In particular, the testing and auditing tools received the lowest scores, suggesting that current tools for general programming practices might need to be fine-tuned to better support cryptographic tasks. In addition, participants mentioned the need for other tools, such as language server protocol (LSP), editor plugin, and domain-specific language (DSL) for programming cryptographic tasks.

Observation 13: *Only 34.8% and 31.9% of participants are satisfied with existing testing tools and audit tools respectively, demonstrating a lack of tools to test and secure cryptographic practices.*

6 DISCUSSION

In this section, we discuss the implications of our findings and point out future directions to improve the Ethereum crypto APIs, facilitate the security of cryptographic practices, and improve third-party templates and tools.

6.1 Improving Ethereum Crypto APIs

Our survey results indicate the need to improve the functionality and usability of current Ethereum APIs.

Usability. Previous studies on traditional cryptographic libraries [63, 72] highlighted that it is essential to hide unnecessary underlying details and provide high-level interfaces for cryptographic tasks. For example, Java Cryptography Architecture [97] provides ready-to-use APIs for high-level cryptographic tasks. Developers can use statements like *Signature.getInstance("SHA256-withECDSA")* to implement signatures in Java, without operating low-level cryptographic operations themselves. To alleviate developers from managing low-level operations, future work could focus on **supporting built-in cryptographic libraries in Solidity**. In fact, four crypto APIs (`KECCAK256`, `ECRECOVER`, `SHA256`, and `RIPEMD160`) have already become built-in functions in Solidity. Developers can use keywords like "*ecrecover*" to invoke these functions, and the Solidity compiler will automatically transform them into calls to the corresponding crypto APIs. However, the challenging task of composing these low-level APIs to accomplish higher-level tasks still falls on developers, requiring further exploration and improvement.

Functionality. Our studies demonstrate that the limited functionalities of current APIs hindered the applications of several promising tasks, *e.g.*, recursive zero-knowledge proofs (*Observation 10*). Notably, in traditional software, developers have the flexibility to explore alternative libraries or even implement missing functionality themselves. However, in smart contracts, the high gas cost of

cryptographic operations makes these system-level crypto APIs the most practical option for developers. Therefore, to avoid fundamentally impeding the application of emerging cryptographic tasks, it is necessary to introduce new crypto APIs required by real-world practices.

Crypto operations mentioned by our real-world survey participants, such as paring-friendly elliptic curves could be promising candidates for improving the functionality of Ethereum cryptographic APIs. In addition to our survey, there is other real-world evidence supporting the inclusion of these candidates. For example, Go-Ethereum [90] have incorporated BLS12-381 [6] into EVM for testing purpose, and several Solidity libraries [38, 40] have implemented elliptic curves like Pasta [49], even with impractical gas cost. However, given the high cost to add precompiled contracts and the backward compatibility requirements, the addition of these candidates might still need to be further evaluated within more detailed contexts. Future work could put more efforts into **collecting real-world evidences of adding these candidates**, *e.g.*, conducting empirical research on the real-world requirements of Ethereum cryptographic APIs.

6.2 The Security of Cryptographic Practices

The security practices for on-chain cryptographic tasks are still in the early stages of development, with deficiencies in both knowledge and tools. For example, while digital signatures have become a common cryptographic task in smart contracts, 37.7% of developers are unfamiliar with potential vulnerabilities related to signature replay attacks (SWC-121). The insecure practices caused by such insufficient understanding have caused real-world security issues [5]. Therefore, it is necessary to **inform developers about the security implications behind Ethereum crypto APIs** and **provide guidelines to prevent common misuses**.

Furthermore, compared to extensive research [105] focusing on traditional cryptographic libraries, the security of Ethereum crypto APIs has not attracted enough attention. While crypto API misuses have been established as common causes of security vulnerabilities [27, 44, 105], what crypto API misuse exists in real-world Ethereum cryptographic practices still lacks systematic analysis. In addition, existing security analysis tools for smart contracts also lack support for crypto-specific vulnerabilities. For example, a recent work [106] conducted a literature review of papers published on top-tier Software Engineering, Security, and Programming Language venues from 2017 to 2022 and documented 17 categories of vulnerabilities that can be detected by existing security tools. However, only the "*weak sources of randomness*" among them is related to cryptographic practices, indicating a lack of tools to detect other crypto-specific vulnerabilities, such as those listed in the SWC list [76]. Future studies could focus on **characterizing Ethereum crypto API misuses in real-world practices** and **incorporating crypto-specific vulnerabilities into existing security analysis tools**.

6.3 Templates and Tools for Implementing Cryptographic Tasks

The complexity of cryptographic implementations often requires developers to rely on templates and tools to fulfill their development needs. For example, 95.4% of the survey participants use templates to accomplish some or all of their cryptographic tasks. However, the feedback from the participants suggests a need for improvement in terms of the usability and functionality of existing templates. For example, OpenZeppelin [68], one of the most popular template libraries, only provides templates for ECDSA signatures and Merkle proofs, while lacking support for other tasks, such as the zero-knowledge proof mentioned by participants. Furthermore, 32.8% of the survey participants encountered difficulties in using templates. Future studies could focus on **improving the usability of existing templates** and **introducing new templates to cover emerging tasks**. Furthermore, several previous studies [88, 89] and industrial solutions [10, 26] have explored using domain-specific languages and code generation tools to automate the code generation process for zero-knowledge proofs. It could be valuable to further **explore the application of code generation tools** for other on-chain cryptographic tasks.

7 THREATS TO VALIDITY

7.1 Internal Validity

Section 3 and Section 4 relied on manual analysis of smart contracts and StackExchange posts, respectively. Therefore, these analysis results might be inherently subjective. To mitigate the subjective biases and strengthen the reliability of our results, we implemented a dual-author approach to every analysis, ensuring that any disagreements were thoroughly resolved. Notably, all authors involved in these tasks have more than three years of experience in smart contract research. In addition, due to the large datasets and time-consuming analysis process, we utilized random sampling to make manual analysis feasible. To mitigate potential sampling bias, the sampling size and ratio for each study are meticulously determined, based on an extensive analysis of previous studies [20, 63, 103] that examined analogous data sources. We also published the labeled dataset and results in our online supplement materials to facilitate replication or further analysis.

Since our study focused on a relatively specific topic, (cryptographic practices on Ethereum), the survey participants in Section 5 might have insufficient understanding of the questions, which could lead to irrational responses. To reduce the impact, we included an "I don't understand" option and excluded such responses from analysis. We also explicitly requested developers to skip questions they found incomprehensible. Several questions in our survey use subjective words (*e.g.*, the term "more challenging" in Q_7) to explore participants' perceptions, which could potentially influence the responses. To mitigate the impact of such suggestiveness on our results, we have implemented several measures, including asking for the rationales behind participants' choices and offering additional choices beyond a simple "*Yes*" or "*No*" to avoid forced decision-making. For example, in Q_7, participants who chose "*Yes*" or "*They are basically the same, but lie in different aspects*" were requested to explicitly explain their choices in Q_8. 39 out of 45 (86.7%) of participants provided detailed rationales, which potentially validates their initial responses to Q_7. Additionally, we adopted a non-probabilistic participant recruitment process for the survey. Although it may result in a pool that is not fully representative of all practitioners, we have tried our best to cover the whole population. Our survey

included participants from 21 countries, all with varied roles and experiences.

7.2 External Validity

Solidity is an ever-evolving language, with new crypto APIs introduced through periodic hard-forks [35]. For example, the Cancun upgrade [9] will add a new precompiled contract [94] to verify KZG proofs [55] in smart contracts. Although our current results cannot cover crypto APIs introduced in the future, our method, findings, and datasets can be extended with minor efforts to incorporate them.

8 RELATED WORK

8.1 General Programming Practices in Smart Contracts

The rapid development of smart contracts has motivated a substantial body of empirical studies on programming practices in smart contracts. They covered the implementation [15, 58, 59, 107], security [20, 95], and many other aspects [2, 8] of programming practices. For example, Bosu *et al.* [15] explored the differences between the implementation of smart contracts and traditional software. They found that 93% of developers believe that smart contract development differs from traditional software development, primarily due to its higher emphasis on security and reliability. Chen *et al.* [21] studied smart contract maintenance-related concerns based on the survey feedback and literature review. Wan *et al.* [95] conducted interviews and surveys to investigate how real-world practitioners build security into smart contract development. They found that 85% and 69% of the practitioners recognize the importance of security and privacy in smart contracts respectively. They commonly re-use templates and employ security tools to enhance the security. However, to the best of our knowledge, these studies mainly focused on general programming tasks, while lack analysis of crypto-specific practices.

8.2 Cryptographic Practices in Traditional Software

Several studies [27, 47, 63] have been conducted to characterize cryptographic practices in traditional software. Hazhirpasand *et al.* [47] collected 489 Github Java projects using Java Cryptography Architecture (JCA) and analyzed the uses and misuses of APIs in them. They found that 99.8% projects using JCA APIs contain at least one misuse. Nadi *et al.* [63] found that JCA is the most used cryptographic library, and securing connections, authenticating user logins, and encrypting files are the top three tasks developers need. Since Ethereum crypto APIs are different from traditional crypto libraries in both design and usage, the results of these studies might not be applicable to Ethereum. Notably, while the majority of our results (*Observation 1-3, 10-13*) are specific to the Ethereum context, some observations such as the categorization of obstacles in *Observation 4-9*, might also be generally applicable. They can validate previous results and provide new evidence of their relevance within the Ethereum context.

8.3 The Usability of Crypto APIs

Several previous studies have focused on the usability of crypto APIs to mitigate the obstacles developers face. Green and Smith [42] proposed ten principles for creating usable and secure crypto APIs, *e.g., integrate crypto functionality into APIs so regular developers don't have to interact with crypto APIs in the first place.*. These principles have been used in several following studies [1, 61, 72]. Patnaik *et al.* [72] identified 16 usability issues of seven existing crypto libraries by analyzing StackOverflow questions and provided evidence to validate Green and Smith's heuristics [42]. Nadi *et al.* [63] analyze why java developer struggle with crypto APIs by analyzing StackOverflow questions and surveying developers. Acar *et al.* [1] conducted a controlled experiment with Python developers to evaluate the usability of five Python cryptographic libraries. They found that libraries designed for simplicity can reduce the decision space of developers and offer security benefits. They also suggested ensuring support for common tasks and providing accessible documentation to developers. Kai *et al.* [61] identified seven major cryptographic libraries in Rust based on a search on Github and other sources, and examined their usability through controlled experiments. Different from these previous studies, we focus on the usability of crypto APIs in the specific context of Ethereum smart contracts. Notably, we not only analyzed usability issues from a single information source, such as StackExchange posts, but also conducted a survey with smart contract practitioners to validate our findings and explore their perceptions.

9 CONCLUSION

We conducted the first empirical study on cryptographic practices in Ethereum smart contracts, through examining 91,484,856 Ethereum transactions and 500 smart contracts, 483 StackExchange posts, and survey input from 78 smart contract practitioners. Our results showed that while Ethereum crypto APIs enable prevalent and diverse on-chain cryptographic tasks, they also pose obstacles for developers. We identified five categories of obstacles from StackExchange posts and conducted an online survey to gain insights from practitioners. Our findings revealed that 57.8% of practitioners encounter obstacles in identifying the detailed steps to implement specific tasks, and 56.3% face difficulties in evaluating the security of their implementations. The feedback from the participants highlighted the gap between low-level crypto APIs and high-level cryptographic tasks and demonstrated the need for improved API functionality and usability, task-based templates, and effective assistance tools. Based on these findings, we provided practical implications to the API designers and template/tool providers and outlined possible directions for future research.

ACKNOWLEDGEMENT

This work is supported by the National Key Research and Development Program of China (2020YFB1005404) and the National Natural Science Foundation of China (62302534, 62332004, 62202011). We also thank the anonymous reviewers of ICSE 2024 for their valuable feedback.

REFERENCES

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

[2] Nemitari Ajienka, Peter Vangorp, and Andrea Capiluppi. 2020. An empirical analysis of source code metrics and smart contract resource consumption. *Journal of Software: Evolution and Process* 32, 10 (2020), e2267.

[3] Vlasov Alex, Olson Kelly, and Stokes Alex. 2020. EIP-1895: Support for an Elliptic Curve Cycle. Retrieved June 17, 2023 from https://eips.ethereum.org/EIPS/eip-1895

[4] Anonymous. 2023. Online Supplement Material. https://zenodo.org/records/10074040

[5] Zhenxuan Bai. 2018. You may pay more than you can imagine. https://github.com/nkbai/defcon26/tree/master/docs

[6] Paulo SLM Barreto, Ben Lynn, and Michael Scott. 2003. Constructing elliptic curves with prescribed embedding degrees. In *Security in Communication Networks: Third International Conference, SCN 2002 Amalfi, Italy, September 11–13, 2002 Revised Papers 3*. Springer, 257–267.

[7] Paulo SLM Barreto and Michael Naehrig. 2005. Pairing-friendly elliptic curves of prime order. In *International workshop on selected areas in cryptography*. Springer, 319–331.

[8] Massimo Bartoletti and Livio Pompianu. 2017. An empirical analysis of smart contracts: platforms, applications, and design patterns. In *Financial Cryptography and Data Security: FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers 21*. Springer, 494–509.

[9] Tim Beiko. 2023. Cancun Network Upgrade Meta Thread. Retrieved June 17, 2023 from https://ethereum-magicians.org/t/cancun-network-upgrade-meta-thread/12060/1

[10] Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina. 2022. Circom: A Circuit Description Language for Building Zero-knowledge Applications. *IEEE Transactions on Dependable and Secure Computing* (2022).

[11] Alexandre Belling. 2018. EIP-1895: Support for an Elliptic Curve Cycle. Retrieved June 17, 2023 from https://eips.ethereum.org/EIPS/eip-1895

[12] Remco Bloemen, Leonid Logvinov, and Jacob Evans. 2017. EIP-712: Typed structured data hashing and signing. Retrieved June 17, 2023 from https://eips.ethereum.org/EIPS/eip-712

[13] Dan Boneh, Xavier Boyen, and Hovav Shacham. 2004. Short Group Signatures. In *Advances in Cryptology – CRYPTO 2004*, Matt Franklin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 41–55.

[14] Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short signatures from the Weil pairing. In *Advances in Cryptology—ASIACRYPT 2001: 7th International Conference on the Theory and Application of Cryptology and Information Security Gold Coast, Australia, December 9–13, 2001 Proceedings 7*. Springer, 514–532.

[15] Amiangshu Bosu, Anindya Iqbal, Rifat Shahriyar, and Partha Chakraborty. 2019. Understanding the motivations, challenges and needs of blockchain software developers: A survey. *Empirical Software Engineering* 24, 4 (2019), 2636–2673.

[16] Vitalik Buterin. 2017. EIP-198: Big integer modular exponentiation. Retrieved June 17, 2023 from https://eips.ethereum.org/EIPS/eip-198

[17] Vitalik Buterin. 2018. EIP-1014: Skinny CREATE2. Retrieved June 17, 2023 from https://eips.ethereum.org/EIPS/eip-1014

[18] Vitalik Buterin and Christian Reitwiessner. 2017. EIP-197: Precompiled contracts for optimal ate pairing check on the elliptic curve alt_bn128. Retrieved June 17, 2023 from https://eips.ethereum.org/EIPS/eip-197

[19] Pascal Caversaccio. 2023. EIP-7266: Remove BLAKE2 compression precompile. Retrieved Oct 17, 2023 from https://eips.ethereum.org/EIPS/eip-7266

[20] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2020. Defining smart contract defects on ethereum. *IEEE Transactions on Software Engineering* 48, 1 (2020), 327–345.

[21] Jiachi Chen, Xin Xia, David Lo, John Grundy, and Xiaohu Yang. 2021. Maintenance-related concerns for post-deployed Ethereum smart contract development: issues, techniques, and future challenges. *Empirical Software Engineering* 26, 6 (2021), 117.

[22] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[23] Ethereum community. 2023. Zero-knowledge Roolups. Retrieved June 17, 2023 from https://ethereum.org/en/developers/docs/scaling/zk-rollups/

[24] Ivan Damgård. 1998. Commitment schemes and zero-knowledge protocols. In *School organized by the European Educational Forum*. Springer, 63–86.

[25] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. 1996. RIPEMD-160: A strengthened version of RIPEMD. In *Fast Software Encryption: Third International Workshop Cambridge, UK, February 21–23 1996 Proceedings 3*. Springer, 71–82.

[26] Jacob Eberhardt and Stefan Tai. 2018. Zokrates-scalable privacy-preserving off-chain computations. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE, 1084–1091.

[27] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. 73–84.

[28] Etherscan. 2023. The Ethereum Blockchain Explorer. Retrieved June 17, 2023 from https://etherscan.io/

[29] Stack Exchange. 2023. Stack Exchange. Retrieved June 17, 2023 from https://stackexchange.com

[30] Stack Exchange. 2023. Stack Exchange API. Retrieved June 17, 2023 from https://api.stackexchange.com/docs

[31] Ethereum Foundation. 2015. Frontier is coming - what to expect, and how to prepare. Retrieved June 17, 2023 from https://blog.ethereum.org/2015/07/22/frontier-is-coming-what-to-expect-and-how-to-prepare

[32] Ethereum Foundation. 2017. Byzantium HF Announcement. Retrieved June 17, 2023 from https://blog.ethereum.org/2017/10/12/byzantium-hf-announcement

[33] Ethereum Foundation. 2019. Ethereum Istanbul Upgrade Announcement. Retrieved June 17, 2023 from https://blog.ethereum.org/2019/11/20/ethereum-istanbul-upgrade-announcement

[34] Ethereum Foundation. 2019. Ethereum Istanbul Upgrade Announcement. Retrieved June 17, 2023 from https://blog.ethereum.org/2019/11/20/ethereum-istanbul-upgrade-announcement

[35] Ethereum Foundation. 2023. The history of Ethereum. Retrieved June 17, 2023 from https://ethereum.org/en/history

[36] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. 2019. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive* (2019).

[37] Manuela Rozalia Gabor et al. 2007. Types of non-probabilistic sampling used in marketing research.„Snowball" sampling. *Management & Marketing-Bucharest* 3 (2007), 80–90.

[38] Github. 2019. Solidity BN256G2. Retrieved Oct 17, 2023 from https://github.com/musalbas/solidity-BN256G2

[39] Github. 2023. GitHub REST API documentation. Retrieved June 17, 2023 from https://docs.github.com/en/rest

[40] Github. 2023. Pasta curves in solidity. Retrieved Oct 17, 2023 from https://github.com/zhenfeizhang/pasta-solidity

[41] Go-Ethereum. 2023. debug Namespace. Retrieved June 17, 2023 from https://geth.ethereum.org/docs/interacting-with-geth/rpc/ns-debug

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

[43] Jens Groth. 2016. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*. Springer, 305–326.

[44] Zuxing Gu, Jiecheng Wu, Jiaxiang Liu, Min Zhou, and Ming Gu. 2019. An empirical study on api-misuse bugs in open-source c programs. In *2019 IEEE 43rd annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 11–20.

[45] Bertoni Guido, Daemen Joan, Peeters Michal, and Gilles Van Assche. 2011. The KECCAK SHA-3 submission. Retrieved June 17, 2023 from https://keccak.team/files/Keccak-submission-3.pdf

[46] Kobi Gurkan. 2022. Optimized BLS multisignatures on EVM. Retrieved June 17, 2023 from https://geometry.xyz/notebook/Optimized-BLS-multisignatures-on-EVM

[47] Mohammadreza Hazhirpasand, Mohammad Ghafari, and Oscar Nierstrasz. 2020. Java cryptography uses in the wild. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–6.

[48] David Holt, AJ Scott, and PD Ewings. 1980. Chi-squared tests with survey data. *Journal of the Royal Statistical Society: Series A (General)* 143, 3 (1980), 303–320.

[49] Daira Hopwood. 2020. The Pasta Curves for Halo 2 and Beyond. Retrieved June 17, 2023 from https://electriccoin.co/blog/the-pasta-curves-for-halo-2-and-beyond

[50] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. 2016. Zcash protocol specification. *GitHub: San Francisco, CA, USA* 4 (2016), 220.

[51] Nikolay Ivanov, Qiben Yan, and Anurag Kompalli. 2023. TxT: Real-time Transaction Encapsulation for Ethereum Smart Contracts. *IEEE Transactions on Information Forensics and Security* 18 (2023), 1141–1155.

[52] Don Johnson, Alfred Menezes, and Scott Vanstone. 2001. The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security* 1 (2001), 36–63.

[53] Ankur Joshi, Saket Kale, Satish Chandel, and D Kumar Pal. 2015. Likert scale: Explored and explained. *British Journal of Applied Science & Technology* 7, 4 (2015), 396.

[54] Mudabbir Kaleem and Weidong Shi. 2021. Demystifying pythia: A survey of chainlink oracles usage on ethereum. In *Financial Cryptography and Data Security. FC 2021 International Workshops: CoDecFin, DeFi, VOTING, and WTSC,*

Virtual Event, March 5, 2021, Revised Selected Papers 25. Springer, 115–123.
[55] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. 2010. Constant-size commitments to polynomials and their applications. In *Advances in Cryptology-ASIACRYPT 2010: 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings 16*. Springer, 177–194.
[56] Barbara A Kitchenham and Shari L Pfleeger. 2008. Personal opinion surveys. *Guide to advanced empirical software engineering* (2008), 63–92.
[57] John Kuszmaul. 2019. Verkle trees. (2019).
[58] Zhou Liao, Shuwei Song, Hang Zhu, Xiapu Luo, Zheyuan He, Renkai Jiang, Ting Chen, Jiachi Chen, Tao Zhang, and Xiaosong Zhang. 2022. Large-scale empirical study of inline assembly on 7.6 million ethereum smart contracts. *IEEE Transactions on Software Engineering* 49, 2 (2022), 777–801.
[59] Lu Liu, Lili Wei, Wuqi Zhang, Ming Wen, Yepang Liu, and Shing-Chi Cheung. 2021. Characterizing transaction-reverting statements in ethereum smart contracts. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 630–641.
[60] Martin Lundfal. 2020. ERC-2612: Permit Extension for EIP-20 Signed Approvals. Retrieved June 17, 2023 from https://eips.ethereum.org/EIPS/eip-2612
[61] Kai Mindermann, Philipp Keck, and Stefan Wagner. 2018. How usable are rust cryptography APIs?. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 143–154.
[62] Ilya Mironov. 2006. Collision-resistant no more: Hash-and-sign paradigm revisited. In *Public Key Cryptography-PKC 2006: 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24-26, 2006. Proceedings 9*. Springer, 140–156.
[63] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping through hoops: Why do Java developers struggle with cryptography APIs?. In *Proceedings of the 38th International Conference on Software Engineering*. 935–946.
[64] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review* (2008).
[65] Corporate Nist. 1992. The digital signature standard. *Commun. ACM* 35, 7 (1992), 36–40.
[66] The National Institute of Standards and Technology (NIST). 2012. SHA-3 Selection Announcement. Retrieved June 17, 2023 from https://csrc.nist.gov/CSRC/media/Projects/Hash-Functions/documents/sha-3_selection_announcement.pdf
[67] Openzepplin. 2023. Checking Signatures On-Chain. Retrieved June 17, 2023 from https://docs.openzeppelin.com/contracts/2.x/utilities
[68] Openzepplin. 2023. The standard for secure blockchain applications. Retrieved June 17, 2023 from https://www.openzeppelin.com/
[69] Stack Overflow. 2019. How to decode SHA256 hash value and retrieve data in Solidity. Retrieved June 17, 2023 from https://stackoverflow.com/questions/56124326
[70] Stack Overflow. 2023. Stack Overflow. Retrieved June 17, 2023 from https://stackoverflow.com
[71] Ilker Ozcelik, Sai Medury, Justin Broaddus, and Anthony Skjellum. 2021. An Overview of Cryptographic Accumulators. *arXiv preprint arXiv:2103.04330* (2021).
[72] Nikhil Patnaik, Joseph Hallett, and Awais Rashid. 2019. Usability Smells: An Analysis of Developers' Struggle With Crypto Libraries.. In *SOUPS@ USENIX Security Symposium*.
[73] Wouter Penard and Tim van Werkhoven. 2008. On the secure hash algorithm family. *Cryptography in context* (2008), 1–18.
[74] Valentina Piantadosi, Giovanni Rosa, Davide Placella, Simone Scalabrino, and Rocco Oliveto. 2023. Detecting functional and security-related issues in smart contracts: A systematic literature review. *Software: Practice and Experience* 53, 2 (2023), 465–495.
[75] Bart Preneel. 1994. Cryptographic hash functions. *European Transactions on Telecommunications* 5, 4 (1994), 431–448.
[76] SWC Registry. 2023. Smart Contract Weakness Classification and Test Cases. Retrieved June 17, 2023 from https://swcregistry.io/
[77] SWC Registry. 2023. Weak Sources of Randomness from Chain Attributes. Retrieved June 17, 2023 from https://swcregistry.io/docs/SWC-120
[78] Christian Reitwiessner. 2017. EIP-196: Precompiled contracts for addition and scalar multiplication on the elliptic curve alt_bn128. Retrieved June 17, 2023 from https://eips.ethereum.org/EIPS/eip-196
[79] Ronald L Rivest, Adi Shamir, and Leonard Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* 21, 2 (1978), 120–126.
[80] Martin P Robillard. 2009. What makes APIs hard to learn? Answers from developers. *IEEE software* 26, 6 (2009), 27–34.
[81] Claus-Peter Schnorr. 1990. Efficient identification and signatures for smart cards. In *Advances in Cryptology—CRYPTO'89 Proceedings 9*. Springer, 239–252.
[82] Christoph Sendner, Huili Chen, Hossein Fereidooni, Lukas Petzi, Jan König, Jasper Stang, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. 2023. Smarter Contracts: Detecting Vulnerabilities in Smart Contracts with Deep Transfer Learning.. In *NDSS*.
[83] Kunjian Song, Nedas Matulevicius, Eddie B de Lima Filho, and Lucas C Cordeiro. 2022. ESBMC-solidity: an SMT-based model checker for solidity smart contracts. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 65–69.
[84] Ethereum StackExchange. 2018. How to use ecrecover() and what it is? Retrieved June 17, 2023 from https://ethereum.stackexchange.com/questions/49261
[85] Ethereum StackExchange. 2019. How to do RSA signature verification based on eip-198. Retrieved June 17, 2023 from https://ethereum.stackexchange.com/questions/73744
[86] Ethereum StackExchange. 2021. Using ECDSA.sol to Sign smartcontract. Retrieved June 17, 2023 from https://ethereum.stackexchange.com/questions/112807
[87] Ethereum StackExchange. 2023. Ethereum StackExchange. Retrieved June 17, 2023 from https://ethereum.stackexchange.com
[88] Samuel Steffen, Benjamin Bichsel, Roger Baumgartner, and Martin Vechev. 2022. Zeestar: Private smart contracts by homomorphic encryption and zero-knowledge proofs. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 179–197.
[89] Samuel Steffen, Benjamin Bichsel, Mario Gersbach, Noa Melchior, Petar Tsankov, and Martin Vechev. 2019. zkay: Specifying and enforcing data privacy in smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1759–1776.
[90] Go Ethereum Team. 2023. Go Ethereum. Retrieved Oct 17, 2023 from https://github.com/ethereum/go-ethereum/tree/master
[91] Justin Thaler et al. 2022. Proofs, arguments, and zero-knowledge. *Foundations and Trends® in Privacy and Security* 4, 2–4 (2022), 117–660.
[92] Hess Tjaden, Luongo Matt, Dyraga Piotr, and Hancock James. 2016. EIP-152: Add BLAKE2 compression function 'F' precompile. Retrieved June 17, 2023 from https://eips.ethereum.org/EIPS/eip-152
[93] Pradeep K Tyagi. 1989. The effects of appeals, anonymity, and feedback on mail survey response patterns from salespeople. *Journal of the Academy of Marketing Science* 17 (1989), 235–241.
[94] Buterin Vitalik, Feist Dankrad, Loerakker Diederik, Kadianakis George, Garnett Matt, Taiwo Mofi, and Dietrichs Ansgar. 2022. EIP-4844: Shard Blob Transactions. Retrieved June 17, 2023 from https://eips.ethereum.org/EIPS/eip-4844
[95] Zhiyuan Wan, Xin Xia, David Lo, Jiachi Chen, Xiapu Luo, and Xiaohu Yang. 2021. Smart contract security: a practitioners' perspective. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1410–1422.
[96] Zhipeng Wang, Stefanos Chaliasos, Kaihua Qin, Liyi Zhou, Lifeng Gao, Pascal Berrang, Benjamin Livshits, and Arthur Gervais. 2023. On how zero-knowledge proof blockchain mixers improve, and worsen user privacy. In *Proceedings of the ACM Web Conference 2023*. 2022–2032.
[97] Jason Weiss. 2004. *Java Cryptography Extensions: Practical Guide for Programmers*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
[98] Bitcoin Wiki. 2019. Secp256k1. Retrieved June 17, 2023 from https://en.bitcoin.it/wiki/Secp256k1
[99] Wikipedia. 2023. List of random number generators. Retrieved June 17, 2023 from https://en.wikipedia.org/wiki/List_of_random_number_generators
[100] Wikipedia. 2023. Proof of work. Retrieved June 17, 2023 from https://en.wikipedia.org/wiki/Proof_of_work
[101] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.
[102] Jed R Wood and Larry E Wood. 2008. Card sorting: current practices and beyond. *Journal of Usability Studies* 4, 1 (2008), 1–6.
[103] Shuo Yang, Jiachi Chen, and Zibin Zheng. 2023. Definition and Detection of Defects in NFT Smart Contracts. In *32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*.
[104] Jingjing Zhang, Yongjie Ye, Weigang Wu, and Xiapu Luo. 2021. Boros: Secure and efficient off-blockchain transactions via payment channel hub. *IEEE Transactions on Dependable and Secure Computing* (2021).
[105] Ying Zhang, Md Mahir Asef Kabir, Ya Xiao, Danfeng Yao, and Na Meng. 2022. Automatic Detection of Java Cryptographic API Misuses: Are We There Yet? *IEEE Transactions on Software Engineering* 49, 1 (2022), 288–303.
[106] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. 2023. Demystifying Exploitable Bugs in Smart Contracts. In *45th International Conference on Software Engineering (ICSE)*.
[107] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach Dinh Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. 2019. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering* 47, 10 (2019), 2084–2106.