EtherCloak: Enabling Multi-Level and Customized Privacy on Account-Model Blockchains

Xinyi Luo, *Graduate Student Member, IEEE,* Kaiping Xue, *Senior Member, IEEE*, Zhuo Xu, Mingrui Ai, *Graduate Student Member, IEEE*, Jianan Hong, Xianchao Zhang, Qibin Sun, *Fellow, IEEE*, Jun Lu

***Abstract*— The lack of privacy-preserving capabilities hinders the further development of blockchains and smart contracts. While numerous privacy solutions have been proposed, limitations persist. Firstly, most existing solutions focus on specific privacy protections such as anonymous payments, private data, or multi-party computation tasks. However, these solutions lack a general privacy ability, allowing users to deploy applications with diverse privacy requirements. Secondly, existing solutions have limited customizability, which means users cannot easily customize and adapt the privacy policies according to their specific demands or preferences. In this paper, we present EtherCloak, which adopts trusted execution environments (TEEs) to achieve a general and customizable privacy policy on account model blockchains, enabling users to conceal any on-chain information. To address the security issues caused by the unreliability of the host the TEE runs on, we design the enclave state check and crash recovery mechanisms and employ them in the block generation process. In addition, we propose an access control mechanism for privacy policy management and data query. We prove that EtherCloak offers general and customizable privacy protection with a minimal increase in transaction size (less than triple) and communication overhead (approximately 10%) compared to Ethereum.**

***Index Terms*—Blockchain, Smart Contract, TEE, Privacy Protection, Consensus.**

I. Introduction

With the rapid development of blockchain technology in cryptocurrencies and decentralized applications (DApps), privacy concerns have become increasingly prominent. For cryptocurrency transactions, public blockchains provide pseudonym-level privacy, which is proved to be insecure by the deanonymisation attacks [1] on both Bitcoin and Ethereum. As for DApps, researchers have proposed various solutions for typical applications such as e-voting [2], [3]. However, the lack of universal and system-level solutions hinders the further deployment of smart contracts in sensitive applications, including healthcare, supply chain, data sharing, certificate authorities, or Internet of Things [4]–[7].

A. Blockchain Privacy Demands and Solutions

Blockchain privacy demands are roughly categorized into identity privacy, contract privacy, and state privacy. Identity privacy involves concealing the identities of transaction participants, enabling the implementation of anonymous applications like anonymous payments, e-voting, and auctions. Zerocash [8] introduces a fundamental architecture for cryptocurrency transactions, leveraging Non-Interactive Zero-Knowledge (NIZK) to construct anonymous coins and employing an oblivious Merkle tree to obscure coin transfers, breaking the linkage between sender and recipient. Subsequently, ZEXE [9] and Zapper [10] extend the concept of anonymous coins to programmable anonymous records, facilitating functionalities such as token issuance and decentralized exchanges.

Contract privacy, on the other hand, is more intricate and varies based on the privacy requirements of DApps. Existing solutions mainly address two demands: private contracts and multi-party computation (MPC) contracts. Private contracts involve sensitive data accessible only to authorized parties, and most solutions [9]–[12] typically leverage NIZK proofs to update encrypted data and provide public verification on the blockchain. MPC contracts, on the other hand, enable multiple participants to contribute private inputs to obtain a public result, supporting applications like poker, auctions, and voting. These tasks can be implemented using cryptographic techniques such as multi-party computation [13]–[15] or Homomorphic encryption [16], [17], or through trusted execution environments (TEE) [18]–[21].

State privacy refers to protecting account states, including user balances and data objects stored within contracts. While there is some overlap between contract privacy and state privacy, the former is specific to individual contract accounts, whereas the latter can encompass a broader scope. For example, [20] protects all account states and restricts access solely to TEEs. State privacy also involves managing data access, specifying which data objects are accessible to particular users. Various access control mechanisms, such as access lists [5], role-based access control [22], [23], and attribute-based access control [24], have been introduced to blockchain systems to address these requirements. Certain DApps may entail multiple privacy demands; for instance, an anonymous auction DApp may necessitate identity privacy for anonymity, MPC contract privacy for sealed auction, and state privacy to conceal user balances.

B. Limitations

Despite the advancements in blockchain privacy solutions, certain limitations persist, notably in generality and customizability. Firstly, no one supports a general privacy

X. Luo, K. Xue, Z. Xu, M. Ai, Q. Sun, and J. Lu are with the School of Cyber Science and Technology, University of Science and Technology of China, Hefei, Anhui 230027, China.

K. Xue, X. Zhang, J. Lu are with Key Laboratory of Medical Electronics and Digital Health of Zhejiang Province, Jiaxing University, Jiaxing, Zhejiang 314001, China.

J. Hong is with the School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai 200240, China.

Corresponding Author: K. Xue (kpxue@ustc.edu.cn)

ability that enables DApps to meet different privacy demands. The anonymous coin-based solutions, represented by [8]–[10], support identity privacy and private contracts but have challenges for MPC contracts and other privacy demands. The MPC contracts solutions, including [15]–[21] do not provide identity privacy and data access control. Solutions for data access control ignore other privacy demands. Secondly, existing solutions lack customizability. They do not allow users to flexibly configure and change their privacy policies according to demands or preferences. Third, existing solutions have limitations on functionality, such as cross-contract invocation or access to the on-chain state (see Section II-B).

Notably, generality and customizability cannot be realized by combining existing solutions because they adopt different system and security models. For instance, the identity protection method used by Zapper [10] cannot work in ZeeStar [16] because they are constructed on stateless and account-model blockchains, respectively. Even if we can set multiple security models in one system to support different privacy policies, the system will become cumbersome and redundant.

C. Our Work

We introduce a novel architecture for blockchain privacy protection that emphasizes generality and customizability, named EtherCloak[1]. EtherCloak integrates TEE with the account model and state transition process of Ethereum, enabling multi-level privacy while allowing users and DApps to tailor privacy policies.

In account-model blockchains like Ethereum, each user or contract possesses an account with an account state that includes the account's balance and other data stored within the contract. Users can send transactions to transfer assets or invoke contracts, resulting in a state transition for the involved accounts. The account model and its transition process inspire a straightforward concept: maintaining different parts of account states and executing related transactions in TEE realizes various privacy policies. For instance, concealing account states and encrypting account addresses in transactions achieves identity privacy, while concealing data objects in contract storage facilitates fine-grained contract privacy.

The novel combination of TEE and blockchain creates a chance for privacy while raising new security issues due to the unreliability of the host running the TEE, breaking correctness, liveness, or availability. First, a corrupted host can load invalid or outdated account states to the enclave, leading to incorrect execution of transactions. Second, a corrupted host can terminate a running enclave, losing confidential data and breaking liveness and data availability. In addition, the flexibility of privacy policy requires access control on policy management and data query, preventing inappropriate data access. Section IV-D gives a detailed explanation of the issues. To conclude, the main contributions are as follows.

- We propose EtherCloak, a blockchain privacy architecture with generality and customization. By combining TEE with the state transition process of account-model blockchains, EtherCloak supports a four-level privacy policy and enables users to configure and adapt their privacy policies based on their demands or preferences.
- Aiming at the correctness and liveness issues caused by the unreliability of the host the TEE runs on, we propose the enclave state check and crash recovery mechanisms. Furthermore, we propose an access control mechanism for policy management and data query. Finally, we take an e-auction DApp as an example to show how EtherCloak enables different privacy demands.
- We prove the security of EtherCloak in terms of consensus security and privacy protection. In addition, we evaluate the block and consensus cost of EtherCloak and compare it with Ethereum to show its practicability. Overall, EtherCloak brings 10% additional consensus cost when half of the transactions are confidential.

The rest of this paper is organized as follows. Section II introduces existing solutions for blockchain privacy protection. Section III explains the fundamental building blocks of Ethereum and Intel SGX that are the cornerstone of our solution. Then we propose EtherCloak in Section IV and V. Section VI shows use cases of EtherCloak, and the security and performance analysis are given in Section VII and VIII. Finally, Section IX concludes our work.

II. Related Work

Regarding blockchain privacy protection, existing solutions focus on identity privacy, contract privacy, and state privacy. Section II-A introduces existing solutions to different privacy requirements. Besides, functionality is also essential to evaluate the practicality of a solution, explained in Section II-B.

A. Privacy

1) Identity Privacy: Identity privacy refers to concealing the involved parties of a transaction, e.g., the sender and recipient of a payment or the caller of a contract invocation. It is a significant requirement for anonymous tasks. Currently, solutions supporting identity privacy are mainly based on anonymous coins or off-chain invocation. The anonymous coin architecture is proposed in Zerocoin [25] that utilizes NIZK and aggregator to implement a decentralized coin-mix protocol, concealing which coin is transferred and the transfer direction. Subsequent solutions, including Zerocash [8], Zexe [9], and Zapper [10], are in line with the basic principle of Zerocoin and make improvement from different aspects such as security, functionality, and usability. Off-chain invocation [19], [26] refers to privately sending transactions to a few selected nodes instead of recording them on-chain. These chosen nodes privately execute transactions and record encrypted results or digests on-chain as evidence. The off-chain invocation leads to a loss of auditability since the invocation behaviors are not recorded on-chain.

2) Private Contract: Private contracts are mainly realized based on NIZK, with a typical architecture in which owners privately update the related data, recording the encrypted data and NIZK proofs on-chain for public verification. Except

[1]The name EtherCloak reflects its ability to provide optional protection for Ethereum-based blockchains, akin to an invisibility cloak for Ethereum

TABLE I
PRIVACY PROTECTION CAPABILITIES OF EXISTING SOLUTIONS

	Zexe,Zapper	zkay	ZeeStar	DeCloak	Ekiden, Pose	L2Chain	[5], [23]	EtherCloak^{ours}
Identity Privacy	Y	N	N	N	Y	N	/	Y
Private Contract	Y	Y	Y	Y	Y	Y	/	Y
MPC Contract	N	N	limited	Y	Y	N	/	Y
State Privacy	stateless	N	N	N	N	Y	/	Y
Data Access	private	private	private	outp-delivery	outp-delivery	TEE only	ACL or RBAC	customizable
Privacy Range	contract	data object	data object	contract	contract	account state	data item	customizable
Cross-Contract	N	Y	Y	N	N	Y	/	Y
Onchain State	stateless	Y	Y	Y	N	Y	/	Y

ZEXE and Zapper, which realize identity privacy and private contracts based on the stateless model of Bitcoin, some studies also realize private contracts on account-model blockchains such as Ethereum. Zether [27] leverages NIZK to construct token contracts. Users can deposit ethers in the contract to acquire an encrypted balance. Subsequently, when they conduct payments by contract, they privately update their balances and construct NIZK proofs to prove the correctness. Unlike Zether's token contracts, BlockMaze [12] designs a dual-balance account model that embeds the NIZK-based private balance to the account state, realizing anonymous payments without dependence on contracts. In line with Zether, Steffen *et al.* [11] presented the zkay language that extends Zether's balance proof to complex logic, enabling contract developers to indicate private values and their owner in the source code.

3) MPC Contract: Some DApps are modeled as multi-party computation (MPC) tasks, where multiple parties upload private inputs and finally generate a public verification output. Verifiability and fairness are two main demands of these MPC DApps. The former requires that all participants can verify the correctness of the output, and the latter requires that all inputs are involved and the output can be delivered to all participants. Earlier solutions [13]–[15] combine MPC technologies with Bitcoin, focusing on fairness and incentive of the computation process. Several solutions, including ZeeStar [16] and SmartFHE [17], leverage functional encryption, such as Homomorphic encryption in the smart contract. Considering the high cost and restricted function of those cryptosystem-based solutions, various solutions leverage the trusted execution environment (TEE) for MPC tasks, including FastKitten [18], Cloak [28], DeCloak [21], Ekiden [19], and Pose [29]. Among them, the former three solutions enable participants to send transactions on-chain, and the TEE acquires transactions and states from the blockchain. In contrast, the latter two solutions adopt the off-chain invocation model. Namely, participants interact with the TEE nodes instead of sending transactions to the blockchain.

4) State Privacy: Some DApps focus on protecting account states, including the balance of users, the data objects stored in contracts, etc. L2Chain [20] encrypts account states and uses TEEs to decrypt states and execute transactions accordingly. Thus, account states are always protected. Besides, some data-oriented DApps may require complex access control on the stored data. Some studies propose data access control schemes for blockchain, such as access list (ACL) [5] or role-based access control (RBAC) [23].

B. Functionality

1) Data Access: Data access indicates who can acquire the protected data or states. Existing solutions fail to combine flexible data access with original properties. The private contract allows only the owner to access the data, while the MPC contract publishes the output. Solutions like [5], [23] support complex access while losing other functions. In addition, lightweight clients should query data from other nodes, and the result should be verifiable, preventing dishonest nodes from returning incorrect results.

2) Protection Range: This property indicates the possible object to be protected. For instance, zkay and ZeeStar only protect several data objects in the contract. Some solutions protect entire contracts based on NIZK (Zexe, Zapper) or by running the contract in TEEs (DeCloak, Ekiden, and Pose). L2Chain protects all account states and conceals execution processes. We require a customizable privacy range, enabling users to decide which part of the account or contract is protected according to their requirements or preferences.

3) Cross-Contract: Many DApps rely on contract libraries (i.e., deployed contracts providing essential functions) or consist of multiple contracts with a mutual invocation, leading cross-contract invocation to be a fundamental requirement. Among existing solutions, zkay and ZeeStar inherit the Ethereum contract functions directly. L2chain also enables cross-contract invocation; however, it requires complex consensus to handle these invocations, leading to higher costs. Other solutions do not support cross-contract invocation.

4) On-chain State: This property denotes the demand to use on-chain account states in a contract, for instance, getting the balance of an account specified in the calldata. On-chain account states (not only the state of one contract) are quite significant for DApps. Among existing solutions, okay, ZeeStar, DeCloak, and L2Chain support on-chain state access. However, in Ekiden and Pose, the contract can only access its own state and receive off-chain input.

TABLE I concludes the privacy and functionality properties of existing solutions and our proposed EtherCloak, and detailed explanations are given here-in-before. In brief, for identity privacy, only Zerocoin-based solutions (Zexe, Zapper) and TEE-based off-chain solutions (Ekiden, Pose) support identity privacy. As for contract privacy, most solutions perform well. For confidential states, only L2Chain provides protection on all account states (while others protect only

specific contract states). As for functionality, most existing solutions do not perform well. We expect a customizable data access and privacy range, enabling users and DApp developers to tailor their privacy policy as needed. In addition, we expect the cross-contract invocation and on-chain state access, as Ethereum supports.

III. Primitives

This section introduces two important primitive technologies of EtherCloak, including Ethereum and Intel SGX.

A. Ethereum

Ethereum [30] is the most representative and widely adopted account-model blockchain. There are two kinds of entities in the Ethereum network, including clients who send transactions and validators who execute transactions and generate blocks. In addition, there are external and contract accounts in Ethereum. An external account has a pair of keys, and a client who owns the private key can send transactions from the account. Contract accounts have no private keys and cannot send transactions. Instead, they are created by validators when executing contract deployment transactions.

As Fig. 1 shows, the account state consists of four fields, i.e., nonce, balance, storageRoot, and codeHash. The latter two fields are empty for external accounts. The storageRoot is a digest of data objects that are stored in the contract, and codeHash is the bytecode of the contract code. The digest is generated based on a Merkle Patricia Trie (MPT). The account state of account a is denoted as $\sigma[a] := \langle \mathsf{nonce}, \mathsf{balance}, \mathsf{storageRoot}, \mathsf{codeHash} \rangle$. All the activated account states are organized into an MPT, called world state, denoted as $\boldsymbol{\sigma}^{\mathsf{rh}} := \mathrm{mpt}(a \to \sigma[a])$, where rh is the roothash of the MPT. MPT is used to verify lightweight clients' queries.

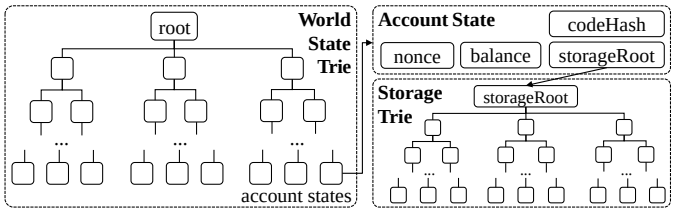


Fig. 1. World state and account state of Ethereum.

For instance, given a client who has rh and a validator who has $\boldsymbol{\sigma}^{\mathsf{rh}}$. The client can send queries to the validator including several accounts $\mathbf{a}=\langle a_1, ..., a_k \rangle$, and the validator will return $\boldsymbol{\sigma}_{\mathbf{a}}=\langle \sigma[a_1], ..., \sigma[a_k] \rangle$ along with a proof π. Then, the client can run an MPT verify function that takes $(\mathbf{a}, \boldsymbol{\sigma}_{\mathbf{a}}, \pi, \mathsf{rh})$ as input to check whether $\boldsymbol{\sigma}_{\mathbf{a}}$ is the correct result acquired from $\boldsymbol{\sigma}^{\mathsf{rh}}$. Similarly, when the client queries the storage data objects in a contract, the validator generates a proof for the account state based on $\boldsymbol{\sigma}^{\mathsf{rh}}$ and then generates a proof for the data object based on the storage tree.

External accounts can send transactions to transfer ethers and deploy or invoke contracts. The transaction contains the transferred value (v), nonce (n) increased by one denoting the number of transactions sent by the sender, sender's signature (s), recipient (r), and invocation input (d). There are also three gas-related fields; we denote them as g^* for brevity. Then, a transaction is denoted as $T := \langle n, s, r, v, d, g^* \rangle$. There are three kinds of transactions, including ether transfer (with empty d), contract invocation (whose r should be a contract account), and contract deployment (with empty r).

For each time slot t (twelve seconds), a specific validator, i.e., the block proposer, selects a group of transactions $\mathbf{T} = \langle T_1, ..., T_n \rangle$ and execute them sequentially, leading to the transition of world states from $\boldsymbol{\sigma}^{\mathsf{rh}_{t-1}}$ to $\boldsymbol{\sigma}^{\mathsf{rh}_t}$, denoted as $\boldsymbol{\sigma}^{\mathsf{rh}_t} = \Gamma(\boldsymbol{\sigma}^{\mathsf{rh}_{t-1}}, \mathbf{T})$. In addition, all transactions in $\mathbf{T}$ should first pass the initial tests of intrinsic validity, including the transaction form, signature, nonce, etc. Then, the proposer generates a new block $B_t = \langle \mathsf{preHash}, \mathsf{rh}_t, \mathbf{T} \rangle$, called a beacon block, and broadcasts it to other validators. Notably, each validator maintains a local blockchain whose last block is B_{t-1}. The validator first check whether $B_t[\mathsf{preHash}]$ equals $\mathrm{hash}(B_{t-1})$. Then, it checks the validity of $\mathbf{T}$ and re-executes $\mathbf{T}$ based on $\boldsymbol{\sigma}^{B_{t-1}[\mathsf{rh}]}$, i.e., the validator's current world state. If the result equals $\boldsymbol{\sigma}^{\mathsf{rh}_t}$, the validator regards the block valid and generates an attestation for it. When a beacon block acquires enough attestations, it upgrades to a justified block. It will upgrade to a finalized block when another justified block links behind it. Afterward, it is permanently accepted by the blockchain unless more than $\frac{2}{3}$ staked ethers are possessed by the adversary. Notably, in practice, the upgrade is executed for every 32 blocks (the final one of which is called a checkpoint) rather than per block.

B. Intel SGX

Intel's Software Guard Extensions (SGX) [31] is a typical instance of the trusted execution environment (TEE), which has been widely used in various privacy protection scenarios [32]. It leverages the trusted hardware to establish a secure container, called the enclave, that provides confidential and trusted execution. Usually, a user encrypts his/her private data and sends the encrypted data and secret key to the host and enclave, respectively. Then, the enclave loads the encrypted data from its host, decrypts data, executes the opcode, encrypts the result, and outputs it to the host along with a remote attestation. Finally, the host returns the output to the user. Intel SGX adopts remote attestation to validate whether an enclave is valid. After remote attestation, users verify whether a message is generated by a valid enclave through the enclave's signature.

IV. System Model and Overview

This section provides the system model and basic workflow of the proposed protocol and analyzes the threat model and our design goals.

A. System Model

Fig. 2 outlines the system model and workflow of the proposed EtherCloak. There are three entities, i.e., clients, validators, and T-validators. Clients are users that generate and broadcast transactions, and validators are responsible

for generating and validating blocks. T-validators refer to blockchain nodes that are equipped with TEE. T-validators facilitate the generation of beacon blocks but do not participate in block consensus. Complying with Ethereum, time is evenly

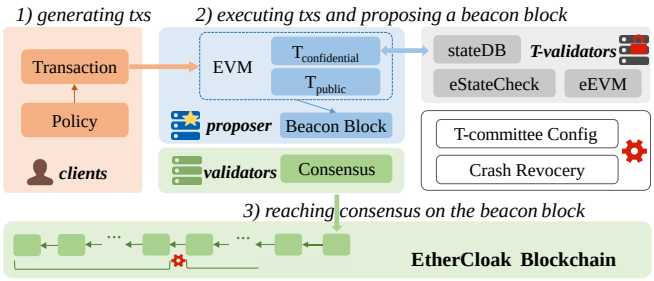


Fig. 2. The system model of EtherCloak.

divided into slots, and multiple slots form an epoch. For each epoch, a T-validator group is selected as the T-committee to execute confidential transactions. The management of the T-committee can be implemented through a smart contract. There are three phases in EtherCloak to generate a block, basically consistent with Ethereum. First, clients generate transactions to transfer assets or deploy or invoke contracts. Second, the proposer collects and executes transactions, updates account states and generates a new block. This phase is assisted by T-validators. Finally, the block is validated and attested by other validators through consensus.

B. Threat model

Clients and validators adopt the same security assumptions as Ethereum, i.e., clients may create invalid transactions for their benefit, and validators may manipulate blocks to record invalid transactions and states on the blockchain. When malicious validators hold less than one-third of the stake, the consensus mechanism of Ethereum can deter these malicious behaviors. Therefore, we focus on T-validators. Each T-validator consists of an enclave that runs on a host. The enclave is assumed to have confidentiality and integrity, meaning the data in the enclave is protected, and the execution of programs is reliable. However, the attack difficulty of hosts is the same as that of a validator, and a corrupted host can manipulate all communications of its enclave or just power off.

In line with Ethereum, we assume an adversary $\mathcal{A}$ can control less than one-third of staked ethers. $\mathcal{A}$ can also create an arbitrary number of accounts; however, it cannot acquire the private key of other accounts. Regarding T-validators, we assume $\mathcal{A}$ can control the hosts of all T-validators in the worst case. According to our proposed crash recovery mechanism that can detect compromised T-validators and displace them, we require that at least one honest T-validator join the system within a limited number of attempts.

C. Requirements

The security requirements of EtherCloak include blockchain security and privacy requirements. Below are the informal descriptions of the security properties, and we formally define them in Section VII.

Requirement 1: correctness, safety, and liveness. As a stateful blockchain, EtherCloak should assure fundamental security properties, including correctness, safety, and liveness. Informally, correctness requires that for each block, the world state is correctly updated based on the state version of its precursor and a set of valid transactions. Safety requires that the blockchain only accepts at most one block at each slot, implying consistency and fork resistance. These two properties ensure that the honest majority validators record the same valid transactions and correct states. Besides, liveness requires that it's always possible to update the world state within a limited time. In EtherCloak, since the world state is divided into public and confidential parts, we extend the liveness requirement to ensure it's always possible to update the world state, both public and confidential, within a limited time.

Requirement 2: privacy and availability. Each account, whether external or contract, is assigned a privacy level that specifies the privacy protection requirement of the account-related information. Privacy requires that an adversary cannot acquire the protected information when the access condition is not satisfied. On the contrary, availability requires that when a client has access authority to specific data, it can obtain the correct results. According to the protected information, there are four privacy levels in EtherCloak. Suppose an account a is involved in transaction T and T is executed and packed into block B. The privacy requirement for different privacy levels is as follows, where $\mathcal{A}$ has the ability defined in Section IV-B. In addition, $\mathcal{A}$ is not the sender of T nor the owner of a. Thus, the four-level privacy is as follows.

- **Public** (level-0).A level-0 account requires no protection.
- **Storage** (level-1). If a is a contract account, it can be set as storage privacy, where a part of data objects (do) on the storage trie are protected, and $\mathcal{A}$ cannot learn their values.
- **State** (level-2). When a has state privacy, $\mathcal{A}$ cannot learn the account state of a, except for the nonce field.
- **Identity** (level-3). When a has identity privacy, $\mathcal{A}$ cannot learn that a is involved in T and B.

Requirement 3: functionality. Enabling the four functionality requirements in Section II-B, including customizable data access control and privacy range, cross-contract invocation, and on-chain state access.

D. Design Overview and Challenges

Based on Ethereum, EtherCloak introduces two design principles. First, each account is assigned a privacy level, including public, storage, state, and identity. As Fig. 3 shows, the world state is divided into a public part $\boldsymbol{\sigma}^{\mathsf{rh}_p}$ and a confidential part $\boldsymbol{\sigma}^{\mathsf{rh}_c}$. Accounts with different privacy levels are protected in different ranges and maintained by different world state parts. Section V-A provides a detailed explanation. Second, the T-validator (TEE) is introduced to maintain protected states and execute corresponding transactions. Assume the proposer collects a group of transactions. It pre-executes the transactions and organizes them as $\mathbf{T} = \langle \mathbf{T}_c, \mathbf{T}_p \rangle$, where transactions in $\mathbf{T}_c$ involve confidential accounts while in $\mathbf{T}_p$ involve only public accounts. Then, the proposer sends

$\mathbf{T}_c$ to the T-validator. The T-validator consists of a host $\mathcal{H}$ and an enclave $\mathcal{E}$. $\mathcal{E}$ runs an EVM called eEVM. $\mathbf{T}_c$ is

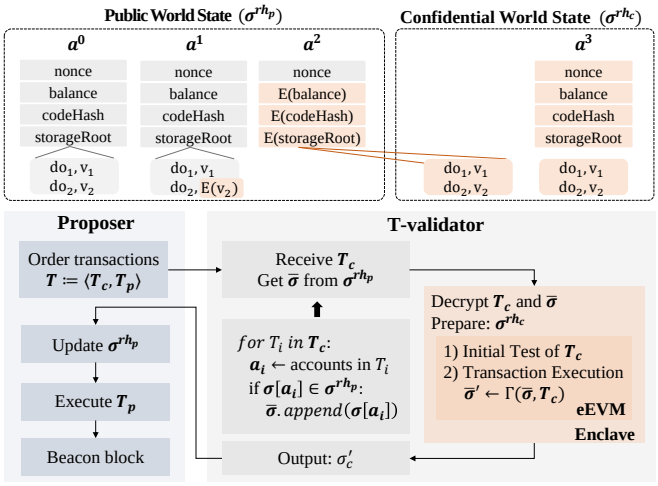


Fig. 3. Design principles: privacy levels and TEE-assisted execution.

received by $\mathcal{H}$. Note that $\mathbf{T}_c$ may involve accounts whose states are maintained on $\boldsymbol{\sigma}^{\mathsf{rh}_p}$. Thus, $\mathcal{H}$ scans $\mathbf{T}_c$ to acquire involved account states (denoted by $\overline{\boldsymbol{\sigma}}$) from $\boldsymbol{\sigma}^{\mathsf{rh}_p}$ and loads $(\mathbf{T}_c, \overline{\boldsymbol{\sigma}})$ into $\mathcal{E}$. Next, $\mathcal{E}$ decrypts transactions and account states, prepares $\boldsymbol{\sigma}^{\mathsf{rh}_c}$, and executes valid transactions, updating $\overline{\boldsymbol{\sigma}}$ to $\overline{\boldsymbol{\sigma}}'$. $\mathcal{E}$ outputs the updated account states $\overline{\boldsymbol{\sigma}}'$ to $\mathcal{H}$ who further sends them to the proposer. The proposer updates $\boldsymbol{\sigma}^{\mathsf{rh}_p}$ according to $\overline{\boldsymbol{\sigma}}'$, executes $\mathbf{T}_p$, and generates the beacon block.

With these two principles, clients and contract developers can customize their privacy policies by choosing a specific privacy level. For a level-1 contract, the protection can be more fine-grained, i.e., protecting specific data objects, such as do_2 of a^2 in Fig 3. In addition, since both the proposer and enclave run an EVM to execute transactions, EtherCloak inherits the cross-contract invocation and on-chain state access functionalities of Ethereum. However, introducing TEE and the adjustable privacy policy brings three issues:

- **Block Verifiability.** The adoption of TEE breaks block verifiability, thus breaking correctness. Although we assume the integrity of the enclave, namely, with correct $\overline{\boldsymbol{\sigma}}$ and $\mathbf{T}_c$, the transition Γ will be correctly executed, and the output $\overline{\boldsymbol{\sigma}}'$ is then correct. However, $\mathcal{E}$ has difficulty in verifying $\overline{\boldsymbol{\sigma}}$ due to lack of communication interface and persistent storage. Thereby, $\mathcal{H}$ can easily break the correctness of $\boldsymbol{\sigma}_c'$ by manipulating $\boldsymbol{\sigma}_c$ and $\mathbf{T}_c$.
- **Crash Recovery.** The usability of TEE is vulnerable since it may run on a dishonest host. Although we can mitigate the problem by letting multiple T-validators run simultaneously, there is still a chance that the hosts of all running T-validators are corrupted, breaking liveness and availability. Thus, a crash recovery mechanism is required to detect the crash and recover the T-validator function. The recovery includes finding an honest T-validator and recovering the lost enclave data such as $\boldsymbol{\sigma}^{\mathsf{rh}_c}$.
- **Access Control.** Policy management and data queries need access control. On one hand, policy management refers to assigning or changing the privacy level of an account. An insecure policy management mechanism can cause a leakage of protected data; for instance, the adversary sets the privacy level of a protected account to the public. On the other hand, data query refers to who can acquire the account state of an account or the value of a data object in a contract. Notably, a lightweight client has to query data from validators or T-validators who may be dishonest. Thus, the query results should be verifiable.

In Section V, we solve these issues by several steps. First, we propose the key components of EtherCloak, especially the confidential account, confidential transaction, and enclave state check mechanism(V-A). On this basis, we introduce the EtherCloak block generation process that supports TEE-assisted transaction execution (V-B). Next, we show the privacy policy management mechanism, including policy assignment and access control (V-C). For clarity, the above protocols are proposed based on the one T-validator setting. Finally, we introduce the T-committee setting and the crash recovery mechanism (V-D).

V. EtherCloak Protocol

A. Components

Several key components are used to construct EtherCloak. First, we introduce the data structure of account states, explaining how to realize the multi-level privacy policy defined in Section IV-C. Second, we show the formation of transactions involved in accounts with different privacy levels. Finally, we propose the enclave state check mechanism to solve the block verifiability issue.

*1) **Confidential Account**:* EtherCloak provides four privacy levels of accounts, including public (level-0), storage (level-1), state (level-2), and identity (level-3), as defined in Section IV-C. Level-1/2/3 accounts are called confidential accounts. TABLE II defines the account state and storage trie of different privacy levels, and an illustration is given in Fig. 3. We write $\sigma[a]$ and $\gamma[a]$ to denote the account state and storage trie, respectively, that is, $\sigma[a] = \langle \mathsf{nonce}, \mathsf{balance}, \mathsf{storageRoot}, \mathsf{codeHash} \rangle$ and $\gamma[a] = \mathrm{mpt}(\mathsf{do}_i, v_i)$. In addition, we assume each account a has a pair of encryption key $(\mathsf{epk}_a, \mathsf{esk}_a)$ used to encrypt account states and use $E_a(\cdot)$ to denote encrypting by epk_a. Notably, $(\mathsf{epk}_a, \mathsf{esk}_a)$ differs from the account key pair used for sending transactions.

TABLE II
Account State and Storage Trie

level	account state	storage trie
0	$\boldsymbol{\sigma}^{\mathsf{rh}_p}[a] := \sigma[a]$	validator: $\gamma[a]$
1	$\boldsymbol{\sigma}^{\mathsf{rh}_p}[a] := \sigma[a]$	validator: $\gamma^1[a]$
2	$\boldsymbol{\sigma}^{\mathsf{rh}_p}[a] := \sigma^2[a])$	enclave: $\gamma[a]$
3	$\boldsymbol{\sigma}^{\mathsf{rh}_c}[a] := \sigma[a]$	enclave: $\gamma[a]$

$\gamma^1[a] = \mathrm{mpt}(\mathsf{do}_i, v_i$ or $E_a(v_i)$
$\sigma^2[a] = \langle \mathsf{nonce}, E_a(\mathsf{balance} \| \mathsf{storageRoot} \| \mathsf{codeHash}) \rangle$

The world state $\boldsymbol{\sigma}^{\mathsf{rh}}$ is divided into a public part $\boldsymbol{\sigma}^{\mathsf{rh}_p}$ and a confidential part $\boldsymbol{\sigma}^{\mathsf{rh}_c}$. $\boldsymbol{\sigma}^{\mathsf{rh}_p}$ is maintained by validators, recording the account state of level-0 to 2 accounts. While

$\boldsymbol{\sigma}^{\mathsf{rh}_c}$ is maintained by the enclave and records level-3 account states. Both $\boldsymbol{\sigma}^{\mathsf{rh}_p}$ and $\boldsymbol{\sigma}^{\mathsf{rh}_c}$ are in the form of MPT. When a is level-0, its account state is in line with Ethereum and is on $\boldsymbol{\sigma}^{\mathsf{rh}_p}$. The storage trie is maintained by validators. When a is level-1, its account state is still $\sigma[a]$, while some data objects on $\gamma[a]$ are protected. When a data object do is protected, the leaf node on $\gamma[a]$ is encrypted, i.e., $E_a(v_i)$. For a level-2 a, the account state is encrypted except for the nonce field. Besides, $\gamma[a]$ is confidentially maintained by the enclave. And when a is level-3, both $\sigma[a]$ and $\gamma[a]$ are confidentially maintained by the enclave, $\sigma[a]$ is on $\boldsymbol{\sigma}^{\mathsf{rh}_c}$.

The public part $\boldsymbol{\sigma}^{\mathsf{rh}_p}$ provides two public interfaces including query and update. Besides, a verification function $\mathrm{verState}$ is used to verify the output of query, as follows.

- $(\boldsymbol{\sigma}_a, \pi) \leftarrow \boldsymbol{\sigma}^{\mathsf{rh}_p}.\mathrm{query}(\mathbf{a})$. This function takes a set of accounts as input, queries the related account states $\boldsymbol{\sigma}_a$ on $\boldsymbol{\sigma}^{\mathsf{rh}_p}$ (i.e., the leaf nodes of the MPT), outputs $\boldsymbol{\sigma}_a$ along with a proof π which is the MPT path proof.
- $\mathsf{rh}' \leftarrow \boldsymbol{\sigma}^{\mathsf{rh}}.\mathrm{update}(\mathbf{a}, \boldsymbol{\sigma}_a)$. This function takes a set of account-state pairs and update the world state $\boldsymbol{\sigma}^{\mathsf{rh}}$. For an existing account, the leaf value on the MPT is updated; and for a non-existing account, a new leaf is inserted.
- $0/1 \leftarrow \mathrm{verState}(\mathbf{a}, \boldsymbol{\sigma}_a, \mathsf{rh}_p, \pi)$. This function takes a set of accounts $\mathbf{a}$, a set of account states $\boldsymbol{\sigma}_a$, a world state digest rh_p and a proof π as inputs, and outputs 1 only when $\boldsymbol{\sigma}_a$ are the account states of $\mathbf{a}$ on world state $\boldsymbol{\sigma}^{\mathsf{rh}_p}$.

The confidential part $\boldsymbol{\sigma}^{\mathsf{rh}_c}$ also provides query interfaces, however, are assisted by the access control mechanism. In addition, the query result of $\boldsymbol{\sigma}^{\mathsf{rh}_p}$ may be encrypted, and one should execute the access control process to acquire the real data. The related algorithm will be given in Section V-C.

*2) **Confidential Transaction**:* A transaction is simplified to $\langle n, s, r, d, v, f\rangle$ that separately denote nonce, signature, recipient, data, value, and flag. When a level-3 account is included, f helps the enclave to identify the account. The formats of transactions are affected by the privacy level of the sender and recipient. Suppose epk_s, epk_r, and epk_e denotes the encryption key of the sender, recipient, and enclave.

The sender's privacy level decides whether to encrypt n and s fields in the transaction. When the sender is level-0 or 2, n and s are not encrypted; and when it is level-3, both are encrypted by epk_s (denoted by $E_s(n\|s)$). The recipient's policy affects r and d. When it is level-3, both r and d are encrypted by epk_r ($E_r(r\|d)$); and when level-2, d is encrypted ($E_r(d)$). Besides, when recipient has level-1 privacy and the calling function requires a protected input, a part of calldata is encrypted. For instance, in the auction function in Section VI, the calldata may be $\mathrm{bid}(E_r(b))$. v is encrypted by epk_r only when both sender and recipient are not level-0 or 1. Finally, when sender or recipient is level-3, f includes the account address encrypted by epk_e. When the enclave executing the transaction, it decrypts f to acquire the account address and then get the corresponding esk to decrypt the transaction. As an example, when both sender and recipient are level-3, the transaction is $\langle E_s(n\|s), E_r(r\|d\|v), f{=}E_e(\mathsf{sender}\|r)\rangle$.

In line with Ethereum [33], the validity of transactions is pre-checked before being included in blocks by the $\mathrm{VerTransaction}$ function. $\mathrm{VerTransaction}$ takes a transaction T as input and outputs 1 only when 1) T is well-constructed, 2) the nonce of T equals the nonce of sender plus 1, and 3) the balance of sender is more than value pluses the basic execution fee. Differently, when sender is a confidential account, the $\mathrm{VerTransaction}$ function should be executed in the enclave.

*3) **Enclave State Check**:* The enclave state check function, denoted by $\mathrm{eStateCheck}$, is executed by $\mathcal{H}$ and $\mathcal{E}$ of the T-validator when executing transactions, enabling $\mathcal{E}$ to check the world state version of $\overline{\boldsymbol{\sigma}}$ loaded by $\mathcal{H}$. It outputs a world state digest, denoted by rh. When $\mathcal{E}$ involves rh in its execution output, other validators can check the world state version on which $\mathcal{E}$ executes the confidential transactions to determine the validity of a new block. Following we show the construction of $\mathrm{eStateCheck}$, the detailed process is shown in Algorithm 1.

Algorithm 1: The construction of $\mathrm{eStateCheck}$

```
   /* Offline Load, run by H                                   */
1  foreach T in T_c do
2      if T.s.level <= 2 then ā.append(T.s) ;
3      if T.r.level <= 2 then
4          ā.append(T.r);
5          if T.r is a contract and T.r.level <=1 then
6              execute T.d, append involved accounts to ā;
7          end
8      end
9  end
10 (σ̄, π) ← σ^{rh_p}.query(ā);
11 load (ā, σ̄, rh_p, π) to E;
   /* Offline Check, run by E                                  */
12 require verState(ā, σ̄, rh_p, π) outputs 1;
13 load (T_c, σ̄‖σ^{rh_c}) to eEVM, start execution;
   /* Online Load and Check                                    */
14 During transaction execution:
15 if encounter an account a not exist in ā then
16     E request σ[a] from H;
17     upon receiving (a, σ[a], rh_p, π) from H, E runs:
18         require verCheck(ā, σ̄, rh_p, π) outputs 1;
19         ā.append(a); σ̄.append(σ[a]);
20 end
21 output rh := rh_c‖rh_p;
```

To execute $\mathbf{T}_c$, $\mathcal{E}$ should acquire all account states that involved by $\mathbf{T}_c$. We write $\overline{\mathbf{a}}$ to denote all involved accounts and $\overline{\boldsymbol{\sigma}}$ to denote their account states. Notably, $\mathbf{T}_c$ may involve accounts with any privacy level, and thus, $\overline{\boldsymbol{\sigma}}$ may involve account states on both $\boldsymbol{\sigma}^{\mathsf{rh}_p}$ and $\boldsymbol{\sigma}^{\mathsf{rh}_c}$. Since $\boldsymbol{\sigma}^{\mathsf{rh}_p}$ cannot be entirely maintained in $\mathcal{E}$ due to memory limitation, the involved account states should be loaded to $\mathcal{E}$ by $\mathcal{H}$ as needed. In addition, for a well-run T-validator, we assume $\mathcal{E}$ maintains the entire $\boldsymbol{\sigma}^{\mathsf{rh}_c}$. However, since $\mathcal{E}$ has no persistent storage, it is necessary to backup $\boldsymbol{\sigma}^{\mathsf{rh}_c}$ in a secure way and reload it when $\mathcal{E}$ restarts. We will discuss this situation in Section V-D.

Upon receiving $\mathbf{T}_c$, $\mathcal{H}$ scans it to acquire an account set $\overline{\mathbf{a}}$ (line 1-11). Specifically, the sender and recipient of each transaction, if not level-3 privacy, are included; if the recipient is a level-0 or level-1 contract account, $\mathcal{H}$ pre-executes the invocation to add other related accounts to $\overline{\mathbf{a}}$.

The pre-execution may be early terminated when encounter confidential accounts. It is adopted to load as much as possible account states during the offline load process. Then, $\mathcal{H}$ acquires $\overline{\boldsymbol{\sigma}}$ and π by executing $\boldsymbol{\sigma}^{\mathsf{rh}}.\mathrm{query}(\mathbf{a})$, and loads $m_1 := (\overline{\mathbf{a}}, \overline{\boldsymbol{\sigma}}, \mathsf{rh}, \pi)$ to $\mathcal{E}$ (line 12-13). Notably, for a contract account, except for the account state, the storage trie should also be loaded to $\mathcal{E}$. To reduce communication overhead, during pre-execution, $\mathcal{H}$ can acquire the involved data objects and only load them to $\mathcal{E}$. The integrity can be proved by MPT path proof. For simplicity, we omit this process in Algorithm 1 and assume $\overline{\boldsymbol{\sigma}}$ contains the required data objects.

When $\mathcal{E}$ receives m_1, it runs $\mathrm{verState}(\overline{\mathbf{a}}, \overline{\boldsymbol{\sigma}}, \mathsf{rh}, \pi)$ and requires output 1. Next, $\mathcal{E}$ starts to execute $\mathbf{T}_c$ based on $\overline{\boldsymbol{\sigma}}$ (line 14-15). The above process is called offline load and check, since it is run before transaction execution. However, since a contract can involve other accounts, during transaction execution, the eEVM may encounter some accounts which are not involved in $\overline{\mathbf{a}}$. In this situation, $\mathcal{E}$ runs the online load and check process to acquire the needed account states, and the state check method is in line with the offline process (line 16-22). Notably, rh_p should be always consistent during the whole execution process of $\mathbf{T}_c$. Finally, $\mathrm{eStateCheck}$ outputs rh that is the state version based on which $\mathcal{E}$ executes $\mathbf{T}_c$.

B. Block Generation

Assume the current slot is t and the proposer is $\mathcal{P}$, it requires four phases to generate and validate a new block B_t, as Fig. 4 shows. For clarity, the protocol is described in the one-T-validator setting, and the T-validator consists of a host $\mathcal{H}$ and an enclave $\mathcal{E}$. We will extend the protocol to the T-committee setting in Section V-D.

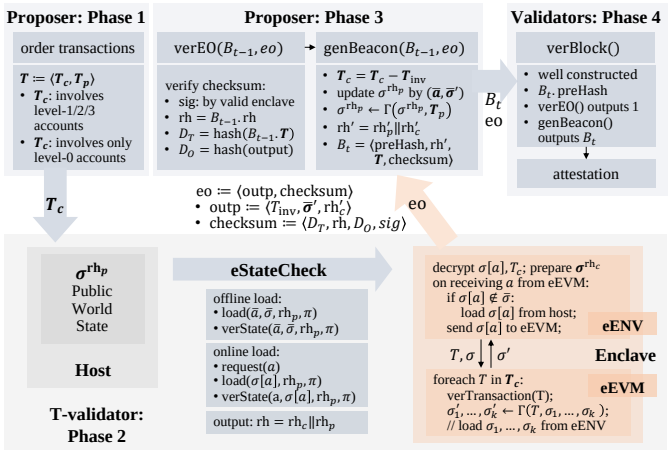


Fig. 4. Block generation.

Phase 1: Proposer Ordering. $\mathcal{P}$ selects a set of transactions and orders them as $\mathbf{T} = \langle T_1, ..., T_n \rangle$. Notably, transactions in $\mathbf{T}$ should pass the initial test including the transaction form, nonce, and signature (for level-0 and level-2 senders). Then, $\mathcal{P}$ pre-executes the transactions to split $\mathbf{T}$ to $\mathbf{T}_c \| \mathbf{T}_p$. $\mathbf{T}_p$ consists of transactions which involved only level-0 accounts, and $\mathbf{T}_c$ consists of the others. Finally, $\mathcal{P}$ sends $\mathbf{T}_c$ to $\mathcal{H}$.

Phase 2: T-validator Execution. According to the enclave state check mechanism ($\mathrm{eStateCheck}$), upon receiving $\mathbf{T}_c$, $\mathcal{H}$ first runs the offline load process to load $(\mathbf{T}_c, \overline{\mathbf{a}}, \overline{\boldsymbol{\sigma}}, \pi, \mathsf{rh})$ into $\mathcal{E}$. We denote the non-eEVM part of $\mathcal{E}$ as the enclave environment (eENV). Subsequently, eENV runs the offline check and then invokes eEVM to execute transactions in $\mathbf{T}_c$ sequentially. eEVM first runs $\mathrm{verTransaction}$. The invalid transactions are recorded in $\mathbf{T}_{\mathsf{inv}}$ and are not executed. During execution, when new account states are needed, eENV runs the online load process to get them from $\mathcal{H}$. Then, the execution process is denoted by $\overline{\boldsymbol{\sigma}}' \| \boldsymbol{\sigma}_c^{\mathsf{rh}'_c} \leftarrow \Gamma(\overline{\boldsymbol{\sigma}} \| \boldsymbol{\sigma}_c^{\mathsf{rh}_c}, \mathbf{T}_c - \mathbf{T}_{\mathsf{inv}})$. The enclave output, denoted by eo, is

$$\mathsf{eo} := \langle \mathsf{outp} = (\mathbf{T}_{\mathsf{inv}}, \overline{\boldsymbol{\sigma}}', \mathsf{rh}'_c), \quad \mathsf{checksum} = (D_T, \mathsf{rh}, \mathrm{hash}(\mathsf{outp}), \mathsf{sig}) \rangle.$$

outp is the execution result including the invalid transactions $\mathbf{T}_{\mathsf{inv}}$, the updated account state $\overline{\boldsymbol{\sigma}}'$ and rh'_c. D_T and rh are used to check the version of $\mathbf{T}_c$ and world state of the enclave execution, where $D_T = \mathrm{hash}(\mathbf{T}_c - \mathbf{T}_{\mathsf{inv}})$ is the digest of valid confidential transactions. sig is a signature used for remote attestation. Finally, $\mathcal{E}$ outputs eo to $\mathcal{H}$ and $\mathcal{H}$ sends eo to P.

Phase 3: Beacon Block. When receiving eo, $\mathcal{P}$ first runs the $\mathrm{verEO}(B_{t-1}.\mathsf{rh}, \mathsf{eo})$ function, where B_{t-1} is the head block of $\mathcal{P}$'s local blockchain. verEO checks the validity of eo from three aspects: sig is generated by a valid $\mathcal{E}$ (remote attestation); $D_T = \mathrm{hash}(\mathbf{T}_c - \mathbf{T}_{\mathsf{inv}})$; and $\mathsf{rh} = B_{t-1}.\mathsf{rh}$. If verEO outputs 1, $\mathcal{P}$ runs $\mathrm{genBeacon}(\boldsymbol{\sigma}^{\mathsf{rh}_p}, \mathbf{T}, \mathsf{eo})$ to generate a beacon block B_t. $\mathrm{genBeacon}$ first sets $\mathbf{T}=\mathbf{T}-\mathbf{T}_{\mathsf{inv}}$, updates $\boldsymbol{\sigma}^{\mathsf{rh}_p}$ according to $\mathbf{a}$ and $\overline{\boldsymbol{\sigma}}'$. Then, it executes $\mathbf{T}_p$, leading to the public world state trie updated to $\boldsymbol{\sigma}^{\mathsf{rh}'_p}$. Finally, $\mathcal{P}$ generates a beacon block $B_t = \langle \mathsf{preHash}, \mathsf{rh}', \mathbf{T}, \mathsf{eo.checksum} \rangle$.

Phase 4: Block Validation. The beacon block is broadcast to other validators for validation. Each validator $\mathcal{V}$ runs the $\mathrm{verBlock}(B_{t-1}, B_t)$ function and generates an attestation on B_t only when the output is 1, where B_{t-1} is the head block of $\mathcal{V}$'s local blockchain. $\mathrm{verBlock}$ checks three items: $B_t.\mathsf{preHash} = \mathrm{hashBlock}(B_{t-1})$; $\mathrm{verEO}(B_{t-1}.\mathsf{rh}, \mathsf{eo})$ outputs 1; $\mathrm{genBeacon}(\boldsymbol{\sigma}^{\mathsf{rh}_p}, \mathbf{T}, \mathsf{eo})$ outputs B_t. When B_t acquires enough attestations, it becomes a justified block, and can further become a finalized block if another justified block is added behind it. This consensus process remains unchanged compared with Ethereum. Considering readability, we omit some details including gas, receipt, event and log, etc. We discuss these components in Section V-E2.

C. Policy Management and Data Query

1) Privacy Policy: Each EtherCloak account a is assigned a policy $\mathsf{policy}[a] := \langle \mathsf{level}, (\mathsf{req}, \mathsf{acp})_i \rangle$. Among them, level is the privacy level, i.e., 0 to 3. When $\mathsf{level} = 1$, a set of data objects that are to be protected should be included. req denotes a possible request and acp is the access control policy of req. A policy may consist of multiple $(\mathsf{req}, \mathsf{acp})$ pairs. An acp consists of an identity policy idp and a condition policy cp, where idp contains a set of account addresses, and cp is a piece of code that can be executed by the enclave and outputs $0/1$. There are two kinds of req, including the privacy level change request $\mathsf{policyReq}$ and data query request $\mathsf{queryReq}$.

- $\mathsf{policyReq}$. $\mathsf{policyReq}$ contains a policy change request and its corresponding acp indicates when the request

TABLE III
TYPES OF policyReq

reqType	input	function
newPolicy	policy$[a]$	replace policy$[a]$ with a new one
newLevel	level	replace level with a new one
newLevel^1	$(\mathsf{do}, 0/1)$	publish/protect do, for level-1 contracts
appACP	$(\mathsf{req}, \mathsf{acp})$	add a new $(\mathsf{req}, \mathsf{acp})$ to policy$[a]$
delACP	req	delete $(\mathsf{req}, \mathsf{acp})$ of a given req

can be executed. A client can send a policy transaction T_{policy} with data=(policyReq‖input) and recipient=a to update policy$[a]$. TABLE III lists the five possible types of policyReq. In policy, req can contain only reqType, implying all possible input. Or it can contain both reqType and input. For instance, $(\mathsf{newLevel}^1, \mathsf{acp})$ indicates that when acp is satisfied, it is allowed to publish or protect any data object. And $(\mathsf{newLevel}^1 \| (\mathsf{do}, 0), \mathsf{acp})$ indicates that only do can be published when satisfying acp.

- queryReq. queryReq has two types, including state and storage, referring to getting the account state or data objects. A $(\mathsf{state}, \mathsf{acp})$ policy specifies the access policy of getting the account state, and a $(\mathsf{storage}, \mathsf{do}, \mathsf{acp})$ policy specifies the access policy of getting the value of do. In addition, when do is default, i.e., $(\mathsf{storage}, \mathsf{acp})$, when acp is satisfied, all data objects are accessible.

All external accounts adopt the same acp, i.e., acp=(a, ϕ) for external account a. Namely, only the owner can change the privacy level and acquire the account state without a condition restriction (cp=ϕ). As a result, policyReq for an external account has only one formation, i.e., $(\mathsf{newLevel}, \mathsf{level})$. For a contract account, its policy is specified in the contract code.

2) Account Creation: Each external account is created as level-0. Once the account has enough balance, it can send a T_{policy} to change its privacy level. A contract account is created by the proposer or enclave when executing the contract deployment transaction. If the initial policy contains level-0 privacy, it is regarded as a public transaction and executed by the proposer; otherwise, it is regarded confidential and executed by the T-validator. In the latter case, when level=3, the account state is added to $\boldsymbol{\sigma}^{\mathsf{rh}_c}$. And when level=1 or 2, the account state is added to $\overline{\boldsymbol{\sigma}}'$ in eo and then added to $\boldsymbol{\sigma}^{\mathsf{rh}_p}$.

3) Policy Transaction: T_{policy} will be included in $\mathbf{T}_c$ and executed by the T-validator. When executing a T_{policy}, $\mathcal{E}$ runs $\mathrm{checkPolicy}(\mathrm{T}_{\mathsf{policy}})$ to check whether the change is allowed. When r is an external account, $\mathrm{checkPolicy}$ outputs 1 only when the sender of T_{policy} is the same as the recipient and policyReq=$(\mathsf{newLevel}, \mathsf{level})$. And when r is a contract, $\mathrm{checkPolicy}$ acquires the corresponding (acp) and outputs 1 only when the sender is in idp and cp is reached. When $\mathrm{checkPolicy}$ outputs 1, $\mathcal{E}$ updates $\sigma[r]$ according to the new privacy level. When changePolicy=3, $\sigma[r]$ is added to $\boldsymbol{\sigma}^{\mathsf{rh}_c}$; otherwise, the updated $\sigma[r]$ is included in $\overline{\boldsymbol{\sigma}}'$ of eo.

4) Key Distribution: When the enclave executes a T_{policy} or executes a contract deployment transaction whose level is not 0, it generates an encryption key pair $(\mathsf{epk}, \mathsf{esk})$ for the account. If the account is external, the enclave sends esk to the sender from a secure channel. And when the account is a level-1 contract with policy queryReq=storage and cp = ϕ, or is a level-2 contract with a policy queryReq=state and cp = ϕ, the enclave sends esk to the clients included in idp. This implies that clients in idp can acquire $\sigma[a]$ anytime. If there's no such policy, esk is kept by the enclave.

5) Data Query: A lightweight client should query data from validators who maintain the world state. There are two query types, including account states and data objects in contracts; the latter is generally implemented through the view function. There are two methods for a client to acquire the protected data, which are adopted by different privacy policies and query types. First, if the encrypted data is maintained on $\boldsymbol{\sigma}^{\mathsf{rh}_p}$, clients who have the corresponding esk can decrypt the data at any time. Second, for data on $\boldsymbol{\sigma}^{\mathsf{rh}_c}$, clients should send query requests to a T-validator. The enclave then checks the corresponding acp and returns the data through a secure channel if satisfied.

- queryType=state. When a is level-$0/1$, any client can acquire $\sigma[a]$ through the $\boldsymbol{\sigma}^{\mathsf{rh}_p}.\mathrm{query}(a)$ interface and verify the result by the $\mathrm{verState}()$ function. When a is level-2, the result of $\boldsymbol{\sigma}^{\mathsf{rh}_p}.\mathrm{query}(a)$ is $\sigma^2[a]$. Only a client with esk_a can decrypt it. And when a is level-3, a client should send a query request to a T-validator. Then, the enclave checks acp of queryType=state in policy$[a]$. If the client is in idp and cp is satisfied, the enclave sends $\sigma[a]$ to the client through a secure channel.
- queryType=storage. The query of data objects is implemented through view functions. When the contract is level-0 or when it is level-1 while the target do is not protected, any node with $\boldsymbol{\sigma}^{\mathsf{rh}_p}$ can return the result. And when do is a protected data object of a level-1 contract, validators only return $E_a(v[\mathsf{do}])$. If the client acquires esk_a during key distribution, it can decrypt the result. Otherwise, the view function should be executed by the enclave. It checks acp of do and returns the value to the client through a secure channel only when acp is satisfied.

D. T-committee and Crash Recovery

To enhance robustness, EtherCloak requires that multiple T-validators work simultaneously, denoted by the T-committee mechanism. As a result, as long as one T-validator in the T-committee has an honest host, an honest proposer can acquire a valid eo and generate a valid block. To this end, EtherCloak maintains a global T-validator candidate list and randomly selects a T-validator group to form the T-committee. Enclaves in a newly formed T-committee negotiate an enclave encryption key pair $(\mathsf{epk}_e, \mathsf{esk}_e)$ and publish epk_e, which will be used to generate the flag f in transactions. In addition, the selection process assigns T-validators a fixed sequence. When multiple T-validators generate different execution results, for instance, assign different encryption key pairs to the same account, only the first T-validator's result is accepted. The T-committee is updated periodically, and we omit the selection process since it has become a regular function.

Crash Detection. However, as we discussed in Section IV-B, there is a chance that all T-validators in the T-committee are

corrupted. In such a case, the proposer and validators cannot acquire a valid eo. Thus, $\mathbf{T}_c$ will be discarded, and the new block only contains $\mathbf{T}_p$. In that case, liveness is breached since $\boldsymbol{\sigma}^{\mathsf{rh}_c}$ will not be updated. In addition, availability is also breached to some extent since some data can only be acquired through T-validators. To solve the problem, we adopt the crash recovery mechanism to guarantee liveness and availability. When $\mathbf{T}_c$ is continuously discarded for multiple blocks (denoted by τ_c), it is regarded that a T-committee crash arises. In that case, a new T-committee will be selected.

Data Recovery. In a normal T-committee update process, enclaves pass the encryption private keys (esk) of confidential accounts and $\boldsymbol{\sigma}^{\mathsf{rh}_c}$ so that new enclaves can access protected data. However, when a crash occurs, the above process cannot work. To address the issue, T-validators who are not in the T-committee are employed to backup data, and these T-validators are called backup T-validators. When an enclave updates esk list when executing T_{policy}, it synchronizes the updated list to storage T-validators. As for $\boldsymbol{\sigma}^{\mathsf{rh}_c}$, storage T-validators can update it by executing $\mathbf{T}_c$ in new blocks. Once a crash occurs, T-validators in the new T-committee can acquire backup data from storage T-validators.

E. Extensions

1) Data Offloading: When invoking a level-2 or level-3 contract, the calldata (i.e., the data field in the transaction) can be directly submitted to the T-validator. And the data field of the transaction can contain only the digest of the calldata. As a result, the transaction size and communication cost can be significantly reduced. When executing the transaction, the enclave need to check whether the calldata is consistent with the digest recorded in the transaction.

2) Omitted Details: The construction of EtherCloak omits several details including gas, receipt, event and log. Since both public and confidential transactions are executed in EVM/eEVM, EtherCloak inherits the processing logic of these components. For instance, EVM/eEVM computes gas during transaction execution and deduct the gas fee from the sender's account, generates a receipt after each transaction, and generates a log when an event is triggered. Only two principles should be noted to implement these components in EtherCloak. First, the gas consumption of confidential transactions should be higher than public transactions, generating a positive incentive for T-validators. We require a future work to determine the exact value of gas. Second, the confidential information in receipt and log should be protected, preventing from privacy leakage through these components.

3) Pegging to Ethereum: To support confidential account, EtherCloak inevitably modifies the data structure and transaction execution process compared to Ethereum. Thus, EtherCloak intrinsically runs as an independent blockchain and is not compatible with legacy blockchains such as Ethereum, leading to a reduction of practicality. To eliminate the incompatibility issue, we can leverage the peg-based sidechain architecture to peg EtherCloak to Ethereum, making EtherCloak works as a layer-2 protocol of Ethereum.

VI. Use Cases

A client who owns an external account can set the privacy level to be 0, 2, 3, meaning no privacy protection, protecting balance, and protecting all behaviors, respectively. As for a contract account, its privacy level is specified by its developer in the contract code. A level-1 contract can be used to implement applications that have both public and confidential parts, enabling customized data access control. Level-2 contracts protect all data and functions. Level-3 accounts are silent on the blockchain: no one else can learn anything about a level-3 account, except that a client can know when it has invoked it. The sender and recipient in a transaction can have different privacy levels. We use a typical e-auction application to show how to implement different privacy demands by combining different privacy levels.

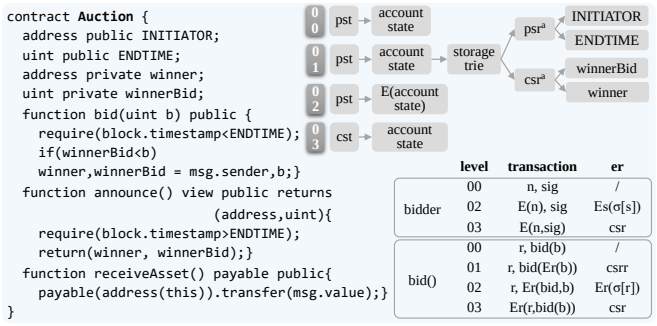


Fig. 5. Essential codes of confidential auction.

Take a simple e-auction whose essential codes are given in Fig. 5 as an example. The auction is implemented as an Auction contract, and bidders bid by invoking the $\mathrm{bid}()$ function to input a value. After the auction ends, participants can invoke the $\mathrm{announce}()$ function to learn the winner, i.e., the bidder with the highest bid. Finally, the winner transfers related assets to the contract through $\mathrm{receiveAsset}()$ function. If the auction is public, e.g., a judicial foreclosure, the contract can be implemented with level-0. If the auction should be partially protected, e.g., the auction information is public while the bid is confidential, it can be with level-1 with protected winner and winnerBid. Bidders invoke $\mathrm{bid}()$ function by encrypting only the bid ($\mathrm{bid}(E_r(b))$). Or it can be directly implemented through a level-2 contract, and bidders should encrypt the entire invocation calldata ($E_r(\mathrm{bid}(b))$). When the initiator wants the full auction process to be protected, the contract can be implemented through a level-3 account, and both r and $\mathrm{bid}(b)$ should be encrypted in the invocation transaction ($E_r(r \| \mathrm{bid}(b))$). Moreover, one contract can be repeatedly used for different auctions with different privacy requirements by changing the privacy level through policy transactions.

From the perspective of bidders, a level-0 bidder is public. Although the bidder can provide an encrypted value during the bid process, the asset transfer will reveal the bid if the bidder is the winner. This makes it risky to allow level-0 users to participate in confidential auctions; that is, if the user is the final winner, it will lead to the disclosure of the final price. For a level-2 bidder, others will learn his/her

participation in the auction. But the amount of transferred asset is protected. A level-3 bidder is completely invisible. Even the auction contract is public, others cannot learn that the bidder participants in the auction. To conclude, the confidentiality of a DApp activity is determined by the privacy level of both the user (i.e., the sender of the transaction) and the DApp contract.

The announce function (i.e., the winner and winnerBid) should be combined with an access control policy according to the privacy requirements of the auction, and there are several ways to realize it.

- *Case 1: sealed auction with published winner.* The auction contract is set to $\mathsf{level}{=}1, \langle \mathsf{winner}, \mathsf{winnerBid} \rangle$, indicating protection of the two data objects. It also contains a $\mathsf{policyReq}{=}\ (\mathsf{newLevel}^1 \| (\mathsf{winner}, 0))$ with $\mathsf{acp} = (\mathsf{idp{:}anyone}, \mathsf{cp{:}time}{>}\tau)$, denoting that winner can be published by anyone when the auction ends (i.e., $\mathsf{time}{>}\tau$). A client can send T_{policy} contains policyReq to publish winner after the auction ends. Later, anyone can invoke the $\mathrm{announce}()$ function to learn the winner.
- *Case 2: sealed auction with protected winner.* Then, the auction contract is set to $\mathsf{level}{=}1, \langle \mathsf{winner}, \mathsf{winnerBid} \rangle$. We add a requirement in the $\mathrm{announce}$ function, namely, $\mathrm{require}(\mathsf{msg.sender}{=}\mathsf{winner})$. Thus, a bidder can invoke $\mathrm{account}$ to know whether he/she is the winner. In addition, bidders should invoke $\mathrm{bid}()$ from level-3 accounts. Otherwise, the $\mathrm{announce}$ and $\mathrm{receiveAsset}$ functions will reveal the winner.
- Case 3: Anonymous auction. When a bidder invokes bid from a level-3 account, he/she becomes anonymous, whatever the privacy policy of the auction contract.

VII. Security Analysis

A. Consensus Security

Consensus security includes safety and liveness. Note that EtherCloak directly adopts the Ethereum Casper consensus protocol, except for block generation and validation processes.

Definition 1. *(Reliable Block Verification) A block verification algorithm* $\mathrm{VerBlock}$ *takes a blockchain head* B_{t-1} *and a new block* B_t *as input and outputs* $0/1$. *A reliable* $\mathrm{VerBlock}$ *outputs* 1 *only when 1)* B_t *is well constructed, 2)* B_t *is an immediate successor of* B_{t-1}*, 3)* $\mathrm{verTransaction}(\mathbf{T})$ *outputs* 1*, and 4)* $\sigma^{B_t.\mathsf{rh}} \leftarrow \Gamma(\sigma^{B_{t-1}.\mathsf{rh}}, B_t[\mathbf{T}])$.

A reliable $\mathrm{VerBlock}$ ensures that an honest validator only provides attestations to a block with valid transactions and correct states. Thus, when $\mathrm{VerBlock}$ is reliable, local correctness is guaranteed.

Lemma 1. *Assume a second pre-image resistant hash function, the* $\mathrm{eStateCheck}$ *algorithm is reliable, namely, it outputs* rh *iff.* $\mathcal{E}$ *executes* $\mathbf{T}_c$ *start from* $\boldsymbol{\sigma}^{\mathsf{rh}}$.

Proof. During the offline load process, $\mathcal{H}$ invokes the query interface of $\boldsymbol{\sigma}^{\mathsf{rh}_p}$ to acquire the offline result. Since $\boldsymbol{\sigma}^{\mathsf{rh}_p}$ is organized as a Merkle Patricia Trie (MPT) and the query interface is a classic key-value query of MPT. Since the hash function adopted by MPT is second pre-image resistant, only when all account states in $\overline{\boldsymbol{\sigma}}$ belong to $\boldsymbol{\sigma}^{\mathsf{rh}_c}$, $\mathrm{verState}$ will output 1. Thus, the offline check is reliable. As for the online load process, $\mathcal{E}$ requires the account state of a only when a is not involved in the offline result. Thus, the online loading result can be regarded to be acquired during offline loading process. As a result, even with online loading, it can be regarded that $\mathcal{E}$ executes $\mathbf{T}_c$ start from $\boldsymbol{\sigma}^{\mathsf{rh}}$. □

Lemma 2. *EtherCloak provides a reliable block verification.*

Proof. Assume an honest validator $\mathcal{V}$ has a blockchain head B_{t-1}. Suppose that $\mathcal{A}$ corrupts the current proposer ($\mathcal{P}$), the host of the T-validator ($\mathcal{H}$), and a client ($\mathcal{C}$). $\mathcal{A}$ constructs a block $B_t{=}\langle \mathsf{preHash}, \mathsf{rh}', \mathbf{T}, \mathsf{eo} \rangle$, where $\mathbf{T}{=}\mathbf{T_c}\|\mathbf{T_p}$ and $\mathsf{eo}{=}\langle D_T, \mathsf{rh}, \overline{\mathbf{a}}\|\overline{\boldsymbol{\sigma}}'\|\mathsf{rh}'_c, sig \rangle$, as defined in Section V-B, and submits it to $\mathcal{V}$. It is insignificant for $\mathcal{A}$ to break the former two check items (i.e., the block construction and preHash) since they can be easily checked. Thus, we focus on $\mathcal{A}$ trying break the third item, denoted by the ErrorTransition game. If $\mathcal{A}$ makes $\mathrm{verBlock}(B_{t-1}, B_t)$ outputs 1 when $\boldsymbol{\sigma}^{B_t.\mathsf{rh}} \neq \Gamma(\boldsymbol{\sigma}^{B_{t-1}.\mathsf{rh}}, B_t.\mathbf{T})$, we claim that $\mathcal{A}$ breaks the reliability of EtherCloak block verification.

Since $\boldsymbol{T}_c$ is executed by $\mathcal{E}$ and $\boldsymbol{T}_p$ is by $\mathcal{P}$, we divide the state transition process into two sub-transitions, i.e.,

$$\boldsymbol{\sigma}_1 \leftarrow \Gamma_{\mathcal{E}}(\boldsymbol{\sigma}_0, \mathbf{T}_c),\ \boldsymbol{\sigma}_2 \leftarrow \Gamma_{\mathcal{P}}(\boldsymbol{\sigma}'_1, \mathbf{T}_p).$$

Thus, $\mathcal{A}$ can succeed only when one of the following cases occurs.

- Case 1: $\boldsymbol{\sigma}_0 \neq \boldsymbol{\sigma}^{B_{t-1}.\mathsf{rh}}$, or $\mathbf{T}_c \neq B_t.\mathbf{T}_c$;
- Case 2: An invalid T is contained in $\mathbf{T}_c$;
- Case 3: $\boldsymbol{\sigma}_1 \neq \Gamma_{\mathcal{E}}(\boldsymbol{\sigma}_0, \mathbf{T}_c)$;
- Case 4: $\boldsymbol{\sigma}'_1 \neq \boldsymbol{\sigma}_1$, or $\mathbf{T}_p \neq B_t.\mathbf{T}_p$;
- Case 5: An invalid T is contained in $\mathbf{T}_p$;
- Case 6: $\boldsymbol{\sigma}_2 \neq \Gamma_{\mathcal{P}}(\boldsymbol{\sigma}'_1, \mathbf{T}_p)$;
- Case 7: $\boldsymbol{\sigma}^{B_t.\mathsf{rh}} \neq \boldsymbol{\sigma}_2$;

Next, we show all the above cases cannot occur, thus $\mathcal{A}$ cannot break the reliability of block verification. For case 1, the verEO function checks $\mathsf{eo.rh}? = B_{t-1}.\mathsf{rh}$ to determine $\boldsymbol{\sigma}_0$ and $\mathrm{hash}(B_t.\mathbf{T}_c)? = \mathsf{eo}.D_T$ to determine $\mathbf{T}_c$. Thus, case 1 occurs only when the hash function used by MPT (according to Lemma 1) or $\mathrm{hash}(\mathbf{T}_c)$ is not secure. For case 2 and case 3, since $\mathrm{verTransaction}(\mathbf{T}_c)$ and $\Gamma_{\mathcal{E}}$ are both executed in the enclave, they occurs only when the integrity of enclave is broken or the remote attestation fails. For case 4, the $\mathrm{genBeacon}$ function updates $\boldsymbol{\sigma}_0$ to $\boldsymbol{\sigma}'_1$ according to $\mathsf{eo}.(\overline{\mathbf{a}}\|\overline{\boldsymbol{\sigma}}'\|\mathsf{rh}'_c)$. Thus, $\boldsymbol{\sigma}'_1 \neq \boldsymbol{\sigma}_1$ only when the integrity of enclave is broken or the remote attestation fails. Besides, since $\mathcal{V}$ executes $\mathrm{verTransaction}(\mathbf{T}_p)$ and $\Gamma_{\mathcal{P}}$ by itself, $\mathbf{T}_p \neq B_t.\mathbf{T}_p$ will not occur, case 4 and case 5 will not occur. Finally, case 7 will not occur since $\mathcal{V}$ can acquire the digest of $\boldsymbol{\sigma}_2$ after executing $\Gamma_{\mathcal{P}}$ and compare it with $B_t.\mathsf{rh}$. □

Theorem 1. *(Safety) EtherCloak inherits the safety of the Ethereum Casper consensus, namely, for each slot t, only one block* B_t *can be finalized.*

Proof. According to Casper [34], as long as more than two-thirds validators obey the two Casper commandments, safety is guaranteed. The Casper commandments are 1) a validator must not publish two distinct votes for the same target height

and 2) a validator must not vote within the span of its other votes. Notably, although EtherCloak adopts different block generation and verification processes, it brings no effects on the Casper commandments, and thus directly inheriting the safety of Casper. □

Theorem 2. *(Correctness) If B_{t-1} and B_t are both finalized, then* $\mathrm{verBlock}(B_{t-1}, B_t)$ *outputs* 1.

Proof. When $\mathrm{verBlock}(B_{t-1}, B_t)$ outputs 0, honest validators will not vote B_t. Thus, B_t has no chance to be finalized. □

Theorem 3. *(Liveness) There exists a limited τ s.t. $B_t.\mathsf{rh}^p \neq B_{t+\tau}.\mathsf{rh}^p$ and $B_t.\mathsf{rh}^c \neq B_{t+\tau}.\mathsf{rh}^c$.*

Proof. First, EtherCloak inherits the liveness of Ethereum Casper consensus since the two Casper commandments are not obeyed (Theorem 1). Thus, each slot t will have a finalized block $B_t = \langle \mathsf{preHash}, \mathsf{rh}', \mathbf{T}, \mathsf{eo} \rangle$. The problem is that if $\mathbf{T}$ is always empty, although there are always blocks to be finalized, the world state remains unchanged. According to the block generation phase, when $\mathcal{A}$ corrupts all T-validators in the T-committee, $\mathcal{A}$ can lead to no valid eo being generated. In that case, $\mathbf{T}_c$ will be discarded by the new blocks. Thus, only $\boldsymbol{\sigma}^{\mathsf{rh}_p}$ will be updated. Or rather, only the state of level-0 accounts will be updated. Thus, liveness is breached.

However, the crash recovery mechanism solves the above problem and guarantees liveness. When no valid eo is generated for τ_c slots straight, the crash recovery mechanism will select a new T-committee. As we assume that at least one honest T-validator can join the T-committee within a limited number of attempts, denoted by k, then we have $\tau = k \cdot \tau_c$. □

B. Privacy and Data Availability

Theorem 4. *(Identity Privacy) Assume an IND-CCA secure $E(\cdot)$ and a one-way hash function $H(\cdot)$, $\mathcal{A}$ cannot win $\mathsf{IDP}_{\mathcal{A}}$ with a non-negligible advantage.*

Algorithm 2: $\mathsf{IDP}_{\mathcal{A}}$

1 Let a_0 and a_1 be two level-3 accounts;
2 Generate $c_r = E_{a_b}(a_b \| d_b)$, $c_f = E_e(a_b)$;
`// E_a(·): defined in Section V-A2`
3 Generate $T := \langle *, c_r, c_f \rangle$, $b \xleftarrow{\$} \{0, 1\}$;
4 Send (T, a_0, a_1) to $\mathcal{A}$;
5 Obtain b' from $\mathcal{A}$, accept iff $b' = b$;

Proof. $\mathcal{A}$ can extract b directly from T, or execute T with the assistance of enclave and extract b from rh_c.

- Case 1: extract from T, namely, given $c_r{=}E_{a_b}(a_b \| d)$ and $c_f{=}E_e(a_b)$, $\mathcal{A}$ guesses b. According to the IND-CCA security of $E(\cdot)$, $\mathcal{A}$ cannot distinguish a_0 and a_1 through c_f. As for c_r, let $m_0{=}a_0 \| d_0$, $m_1{=}a_1 \| d_1$. Then $\mathcal{A}$ has $c_r, (\mathsf{pk}_0, m_0), (\mathsf{pk}_1, m_1)$ to guess whether $c_r{=}E_{\mathsf{pk}_0}(m_0)$ or $c_r{=}E_{\mathsf{pk}_1}(m_1)$, which is also infeasible under the IND-CCA secure assumption.
- Case 2: extract from rh_c, namely, $\mathcal{A}$ can execute T by the enclave and extract b from the outputted rh_c. Note that rh_c is the roothash of a MPT and $(a_0, \sigma[a_0]), (a_1, \sigma[a_1])$ are two key-value pairs stored on the MPT. If a_b is contained in T, after the execution, $\mathsf{rh}'_c = \boldsymbol{\sigma}^{\mathsf{rh}_c}.\mathrm{update}(a_b, \sigma[a_b]')$. Since $\boldsymbol{\sigma}^{\mathsf{rh}_c}$ consists of numerous $(a, \sigma[a])$ pairs where $\sigma[a]$ is unknown to $\mathcal{A}$, it is impossible for $\mathcal{A}$ to distinguish a_0 and a_1 from $(\mathsf{rh}_c, \mathsf{rh}'_c)$ due to the one-way property of hash function.

□

Theorem 5. *(Data Privacy) Assume an IND-CCA secure $E(\cdot)$, a second pre-image resistant hash function $H(\cdot)$, and an EUF-CMA secure signature scheme $S(\cdot)$. When a has a* $(\mathsf{state}, \mathsf{acp})$ *policy, $\mathcal{A}$ cannot learn $\sigma[a]$ when* acp *is not satisfied. When a has a* $(\mathsf{storage}, \mathsf{do}, \mathsf{acp})$ *policy, $\mathcal{A}$ cannot learn $v[\mathsf{do}]$ when* acp *is not satisfied.*

Proof. The access control policy acp consists of an identity policy idp and a condition policy cp, where idp is verified from signatures and cp is a piece of code run by the enclave. When $\mathcal{A}$ does not possess an account which is in idp, according to the security of the signature scheme, $\mathcal{A}$ can neither acquire esk_a during key distribution nor passes the policy check during data query. Furthermore, without esk_a, $\mathcal{A}$ cannot acquire $\sigma[a]$ from $E_a(\sigma[a])$ or acquire $v[\mathsf{do}]$ from $E_a(v[\mathsf{do}])$. Therefore, $\mathcal{A}$ cannot acquire protected data when it has no account included in idp. In addition, assume $\mathcal{A}$ satisfies idp while cp is not satisfied, $\mathcal{A}$ still cannot acquire protected data. This is because when cp is not empty, $\mathcal{A}$ can only query data from an enclave, and the check of cp is reliably executed by the enclave. Thus, $\mathcal{A}$ cannot acquire the protected data when acp is unsatisfied. □

Theorem 6. *(Data Availability) Assume a second pre-image resistant hash function $H(\cdot)$ and an EUF-CMA secure signature scheme $S(\cdot)$. When* acp *is satisfied, a client $\mathcal{C}$ can acquire the correct data from validators or T-validators.*

Proof. $\mathcal{C}$ has two possible ways to acquire data, including getting public data (which might be encrypted) from validators or requesting protected data from T-validators. The target data can be $\sigma[a]$ or $v[\mathsf{do}]$ when a is a contract. When querying $\sigma[a]$ and a is level-0 or level-1, $\mathcal{C}$ can always get $\sigma[a]$ through the $\boldsymbol{\sigma}^{\mathsf{rh}_p}.\mathrm{query}()$ interface and verify the result by $\mathrm{verState}()$, according to the security of hash function. When a is level-2 with policy $(\mathsf{state}, \mathcal{C} \in \mathsf{idp}, \mathsf{cp}{=}\phi)$, $\mathcal{C}$ can acquire esk_a during key distribution if $\mathsf{cp} = \phi$. When $\mathsf{cp} \neq \phi$ or a is level-3, $\mathcal{C}$ can get $\sigma[a]$ from a T-validator when cp outputs 1 and verify the results through remote attestation of TEE. Similarly, $\mathcal{C}$ can verifiably acquire $v[\mathsf{do}]$ when a is level-0, or is level-1 where do is not protected. And when a is level-1 with protected do, $\mathcal{C}$ can acquire $E_a(v[\mathsf{do}])$ for a level-1 a and decrypt it when owning esk_a; otherwise, $\mathcal{C}$ can request $v[\mathsf{do}]$ from a T-validator and verify the results through remote attestation. □

VIII. Performance Analysis

We focus on evaluating the additional overhead caused by EtherCloak's additions compared to Ethereum, mainly including the transaction, enclave output eo, and T-validator execution. In Section VIII-A, we evaluate the data size of

transactions and blocks with different privacy levels. Since the contract invocation can be arbitrarily complex, to make the evaluation concrete, we take the e-auction application given in Fig. 5 as an example. In Section VIII-B, we evaluate the cost of generating a beacon block, mainly including T-validator execution and enclave output.

Processes related to clients and validators are implemented in Golang, and T-validator-related processes are implemented in C++ and Intel SGX. We use the Microsoft Enclave EVM as the eEVM. Addresses and signatures are based on the secp256k1 curve, just the same as Ethereum, and the encryption is implemented by RSA-1024 with PKCS #1 v2.0. Cryptographies are implemented with Openssl and Sgxssl. All the measurements are taken on a machine with an Intel i5-8500 CPU at 3.0 GHz with 16 GB of RAM (representing a client, validator, or T-validator).

A. Transactions and Blocks

Most fields in a transaction have a fixed size, i.e., n, v, and gas-related fields (three fields) are scalar values (supposed as 8 bytes), r is a 20-byte address, s is the signature consisting of $[R][S]$ with 65-byte length. Therefore, the above fields are a total of 125 bytes. However, data is an unlimited-size byte array that contains the invocation command or contract code. This makes it complex to analyze the total size of a transaction. We, therefore, take the auction contract given in Section VI as an example, and the essential codes are shown in Fig. 5. For a public contract, the calldata size of invoking the $\mathrm{bid}()$ function is 36 bytes.

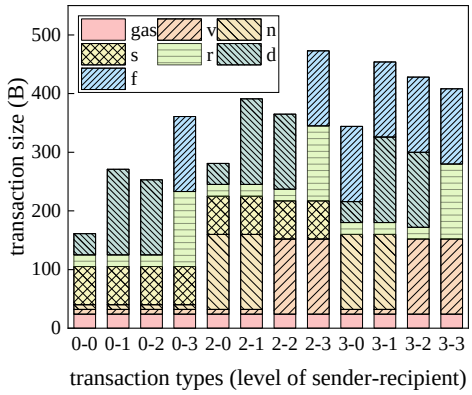


Fig. 6. Size of transactions.

Fig. 6 shows the transaction size with different sender and recipient levels, and data is represented by invoking $\mathrm{bid}()$ function. The 0-0 transaction equals the original Ethereum transaction and is 161 bytes consisting of v, n, s, r, d, and gas-related fields. The transaction size is mainly affected by two factors, i.e., the number of confidential participants and the data. When both participants are confidential, the transaction contains two ciphertexts, while if only one participant is confidential, there's only one ciphertext. For instance, the difference of 253 bytes for 00-02 and 365 bytes for 02-02 is caused by the additional ciphertext. The impact of data is not obvious in the result because the calldata of the auction contract is relatively short (146 bytes for $\mathrm{bid}(E(b))$ and 128 bytes for $E(\mathrm{bid}(b))$). However, when a contract requires a large size of input, it will be useful to adopt the data offloading mechanism, fixing the transaction size to 157 bytes, because the data field becomes a 32-byte hash value.

To conclude, without calldata offloading, the transaction size of EtherCloak is less than triple that of Ethereum, which is a relatively small cost compared with those ZKP-based schemes such as Zexe, with a transaction size around 1000 bytes [9]. And when adopting the calldata offloading mechanism, the transaction size of EtherCloak becomes smaller than Ethereum. Apart from the transaction format, our scheme introduces no additional overhead in the block, except for the eo.checksum, consisting of three hash values and one signature. This is a trivial cost compared to a block. Therefore, the change in block size is consistent with transactions.

B. Transaction Execution and Block Generation

It takes three phases to generate a beacon block, i.e., proposer ordering, T-validator execution, and beacon block generation. To evaluate the performance of the block generation process, according to the average transaction per block value of Ethereum [35], which is 160, we generate 200 transactions for each slot/block with half public and half confidential, all of them are invoking $\mathrm{bid}(b)$. And assume that both public and confidential sets have 20 invalid transactions, the final transaction number in the block is 160.

1) Proposer Ordering: During the proposer ordering phase, the proposer orders received transactions and sends a confidential transaction list ($\mathbf{T}_c$) to each T-validator in the T-committee. The transaction ID is a byte array of length 32, which is the hash of the transaction. Given that the number of confidential transactions is k_c and k_t validators form a T-committee, the communication cost of the proposer is $(k_c k_t + 1) \cdot 32$ bytes where the additional 32 bytes is for the D_T (i.e., $D_T = \mathrm{hash}(\mathbf{T}_c)$). When there are 100 confidential transactions, the size of $\mathbf{T}_c$ is about 3KB.

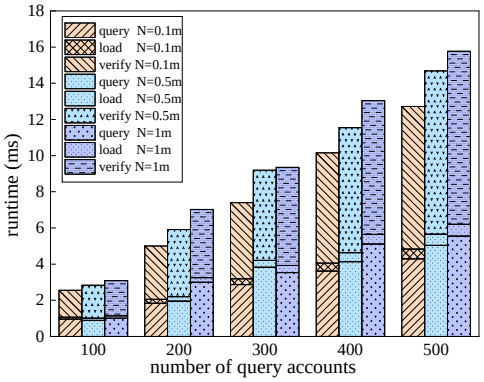


Fig. 7. The runtime of eStateCheck.

2) T-validator Execution: The T-validator execution phase consists of offline state load, decryption of state and transactions, eEVM execution with online state load, and eo output. During offline state load, $\mathcal{H}$ queries required account states through $\boldsymbol{\sigma}^{\mathsf{rh}_P}.\mathrm{query}()$, loading the results and proofs

to $\mathcal{E}$, and $\mathcal{E}$ verifies the results by $\mathrm{verState}()$. The runtime is affected by the total number of accounts on $\boldsymbol{\sigma}^{\mathsf{rh}_p}$ (denoted by N) and the size of $\overline{\mathbf{a}}$ (denoted by n). Roughly, as Fig. 7 shows, the offline state load phase requires only tens of milliseconds even when N reaches one million. After the offline load, $\mathcal{E}$ decrypts account states and transactions and executes transactions in EVM. Fig. 8 shows the runtime of eEVM execution. When there are 200 transactions in $\mathbf{T}_c$, it takes around 1.5s to execute all transactions. Notably, the transaction execution time is affected by the complexity of the invoked contract. In our experiment, all transactions are used to invoke $\mathrm{bid}()$ in the auction contract.

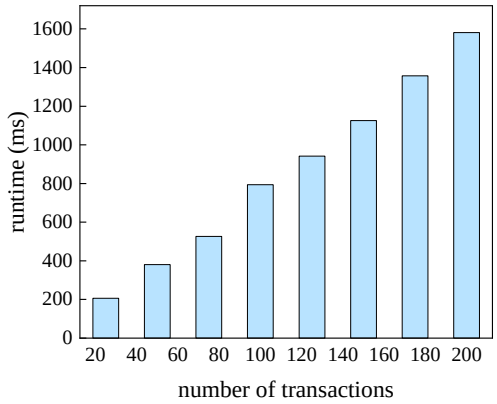


Fig. 8. The runtime of eEVM execution.

According to the above results, the entire T-validator process can be finished within 2s when there are 200 transactions in $\mathbf{T}_c$. This is far less than the average block generation time, i.e., around 10s, meaning that the additional execution cost caused little impact on the consensus process. This result is inevitable because the main consensus bottleneck is generated by the transmission of data over the network. The processing cost on a single machine becomes negligible after using SGX to replace expensive cryptographic algorithms such as zero-knowledge proofs. Therefore, we should pay more attention to whether the new data structures, especially the enclave output eo, will have too much impact on the consensus.

3) Communication Overhead: EtherCloak introduces two impacts on communication overhead compared to Ethereum, including the confidential transactions and enclave output eo. As discussed in Section VIII-A, the size of an EtherCloak transaction is less than triple that of an Ethereum transaction. This section evaluates the communication overhead caused by eo. The enclave output eo consists of outp and checksum. outp contains the invalid transaction list $\mathbf{T}_{\mathsf{inv}}$, the updated account state $\overline{\boldsymbol{\sigma}}'$ and rh'_c. Among them, the size of T_{inv} is $k_i \cdot 32$ bytes where k_i in the number of invalid transactions, rh'_c is a 32-byte hash value. checksum consists of three 32-byte hash values and one 65-byte signature. The most significant part is $\overline{\boldsymbol{\sigma}}'$, including multiple account states and data objects. The account privacy level and invoked function affect the size of $\overline{\boldsymbol{\sigma}}'$. TABLE IV shows the output content of different accounts. For an external account a, when a is level-0, $\sigma[a]$ is included; when level-2, $E_a(\sigma[a])$ is included. For a contract account, when a is level-0, $\sigma[a]$ and the used data objects (winner and winnerBid, simplified to w and wb) are included; when a is level-1, w and wb are encrypted; when a is level-2, $E_a(\sigma[a])$ is included. Notably, level-3 accounts are not included in $\overline{\boldsymbol{\sigma}}'$.

TABLE IV
INCLUDED CONTENTS OF $\overline{\boldsymbol{\sigma}}'$ OF DIFFERENT ACCOUNT LEVELS

type	external		contract		
level	0	2	0	1	2
format	$\sigma[a]$	$E_a(\sigma[a])$	$\sigma[a]\|\|(\mathsf{w}\|\|\mathsf{wb})$	$\sigma[a]\|\|E_a(\mathsf{w}\|\|\mathsf{wb})$	$E_a(\sigma[a])$
size (B)	128	128	180	256	128

The total size of eo represents the additional communication overhead of EtherCloak compared to Ethereum. To evaluate the total size of eo, we create four auction contracts with level-0 to 3 and three external accounts with level-0, 2, and 3, respectively. We let $\mathbf{T}_c$ include 40, 80, 120, and 160 confidential transactions with different combinations of sender-recipient level and evaluate the size of eo. As Fig. 9 shows, the maximum size of eo can reach 50KB when all 160 transactions have level-2 sender and level-1 recipient. And when only level-3 accounts are included, the size of eo is as small as 0.1KB. When $\mathbf{T}_c$ consists of all 11 types of confidential transactions with the same proportion, the size of eo is around 5KB, 10KB, 15KB, and 20KB when there are 40, 80, 120, 160 confidential transactions. To conclude, given that the Ethereum average block size is around 100 KB [35] with about 160 transactions per block, the additional communication overhead of EtherCloak compared to Ethereum is around 5%, 10%, 15%, 20% when the confidential transaction portion is 25%, 50%, 75%, and 100%. It should be noted that the above results are acquired when all transactions are used to invoke the $\mathrm{bid}(b)$ function. If the calldata becomes more complex or the size of involved data objects increases, the communication cost may also increase. However, the data offloading (Section V-E1) mechanism can be adopted to reduce the cost, and the final cost may be smaller than that of Ethereum when the calldata is large.

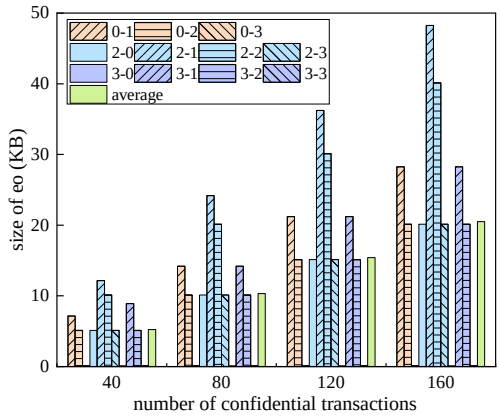


Fig. 9. Size of the enclave output eo.

IX. CONCLUSION AND FUTURE WORK

Aiming at the limitations of existing privacy solutions for blockchain, including lack of generality and customizability,

we proposed EtherCloak, which enables users to conceal any on-chain information according to their demands or willingness. We first proposed the four-level confidential account mechanism. Then, we designed the TEE-assisted block generation protocol, adopting the enclave state check and crash recovery mechanisms to defend against corrupted enclave hosts. We also proposed an access control mechanism to ensure the security of policy management and data query. We discussed practical use cases in a typical confidential auction scenario to show the capability of privacy protection and flexibility. We proved the security of EtherCloak in terms of consensus security, privacy, and data availability. Our performance analysis shows that the transaction size of EtherCloak is less than triple that of Ethereum, and the communication cost is about 10% when half of the transactions are confidential, supporting the practicability.

Limitations and Future Work. Several mechanisms are expected to be supplemented to make the proposed EtherCloak blockchain more practical. Gas should be well-designed to be fair for both T-validators and validators, as well as public and protected users. Besides, the compiler and Ethereum virtual machine (EVM) are expected to be upgraded to make it more convenient for users to write and deploy confidential contracts (especially the level-1 contract). For example, users can simply mark the protected variables or functions and their access policies in Solidity contracts.

ACKNOWLEDGMENT

This work is supported in part by Anhui Province Key Technologies Research & Development Program under Grant No. 2022a05020050, Key Laboratory of Medical Electronics and Digital Health of Zhejiang Province under grant No. MEDH202201, the National Natural Science Foundation of China under Grant No. 62372425, and Youth Innovation Promotion Association of the Chinese Academy of Sciences (CAS) under Grant No. Y202093.

REFERENCES

[1] A. Biryukov, D. Khovratovich, and I. Pustogarov, "Deanonymisation of clients in bitcoin P2P network," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2014, pp. 15–29.

[2] Y. Yang, Z. Guan, Z. Wan, J. Weng, H. H. Pang, and R. H. Deng, "PriScore: Blockchain-based self-tallying election system supporting score voting," *IEEE Transactions on Information Forensics and Security*, vol. 16, no. 1, pp. 4705–4720, 2021.

[3] M. Li, X. Luo, W. Sun, J. Li, and K. Xue, "AvecVoting: Anonymous and verifiable E-voting with untrustworthy counters on blockchain," in *Proceedings of the 2022 IEEE International Conference on Communications (ICC)*. IEEE, 2022, pp. 4751–4756.

[4] R. Henry, A. Herzberg, and A. Kate, "Blockchain access privacy: Challenges and directions," *IEEE Security & Privacy*, vol. 16, no. 4, pp. 38–45, 2018.

[5] J. Xu, K. Xue, S. Li *et al.*, "Healthchain: A blockchain-based privacy preserving scheme for large-scale health data," *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8770–8781, 2019.

[6] X. Luo, K. Xue, J. Li, R. Li, and D. S. Wei, "Make rental reliable: Blockchain-based network slice management framework with SLA guarantee," *IEEE Communications Magazine*, vol. 61, no. 7, pp. 142–148, 2023.

[7] K. Xue, X. Luo, H. Tian *et al.*, "A blockchain based user subscription data management and access control scheme in mobile communication networks," *IEEE Transactions on Vehicular Technology*, vol. 71, no. 3, pp. 3108–3120, 2021.

[8] E. B. Sasson, A. Chiesa, C. Garman *et al.*, "Zerocash: Decentralized anonymous payments from bitcoin," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2015, pp. 459–474.

[9] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu, "ZEXE: Enabling decentralized private computation," in *Proceedings of the 2020 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2020, pp. 947–964.

[10] S. Steffen, B. Bichsel, and M. Vechev, "Zapper: Smart contracts with data and identity privacy," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2022, pp. 2735–2749.

[11] S. Steffen, B. Bichsel, M. Gersbach, N. Melchior, P. Tsankov, and M. Vechev, "Zkay: Specifying and enforcing data privacy in smart contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019, pp. 1759–1776.

[12] Z. Guan, Z. Wan, Y. Yang, Y. Zhou, and B. Huang, "BlockMaze: An efficient privacy-preserving account-model blockchain based on zk-SNARKs," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, pp. 1446–1463, 2022.

[13] R. Kumaresan and I. Bentov, "How to use bitcoin to incentivize correct computations," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2014, pp. 30–41.

[14] R. Kumaresan, T. Moran, and I. Bentov, "How to use bitcoin to play decentralized poker," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Association for Computing Machinery, 2015, pp. 195–206.

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

[16] S. Steffen, B. Bichsel, R. Baumgartner, and M. Vechev, "ZeeStar: Private smart contracts by homomorphic encryption and zero-knowledge proofs," in *Proceedings of the 2022 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2022, pp. 1543–1561.

[17] R. Solomon, R. Weber, and G. Almashaqbeh, "smartFHE: Privacy-preserving smart contracts from fully homomorphic encryption," in *Proceedings of the 8th IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2023, pp. 309–331.

[18] P. Das, L. Eckey, T. Frassetto *et al.*, "FastKitten: Practical smart contracts on bitcoin," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*. USENIX, 2019, pp. 801–818.

[19] R. Cheng, F. Zhang, J. Kos *et al.*, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts," in *Proceedings of the 2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 185–200.

[20] Z. Xu and L. Chen, "L2chain: Towards high-performance, confidential and secure layer-2 blockchain solution for decentralized applications," *Proceedings of the VLDB Endowment*, vol. 16, no. 4, pp. 986–999, 2022.

[21] Q. Ren, Y. Li, Y. Wu *et al.*, "DeCloak: Enable secure and cheap multi-party transactions on legacy blockchains by a minimally trusted TEE network," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 88–103, 2024.

[22] P. Ruan, Y. Kanza, B. C. Ooi, and D. Srivastava, "Ledgerview: Access-control views on hyperledger fabric," in *Proceedings of the 2022 International Conference on Management of Data (SIGMOD)*. ACM, 2022, pp. 2218–2231.

[23] E. Kokoris Kogias, E. C. Alp, L. Gasser *et al.*, "CALYPSO: Private data management for decentralized ledgers," *Proceedings of the VLDB Endowment*, vol. 14, no. 4, pp. 586–599, 2020.

[24] L. Zhang, T. Zhang, Q. Wu, Y. Mu, and F. Rezaeibagha, "Secure decentralized attribute-based sharing of personal health records with blockchain," *IEEE Internet of Things Journal*, vol. 9, no. 14, pp. 12 482–12 496, 2022.

[25] I. Miers, C. Garman, M. Green, and A. D. Rubin, "Zerocoin: Anonymous distributed e-cash from bitcoin," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2013, pp. 397–411.

[26] H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten, "Arbitrum: Scalable, private smart contracts," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*. USENIX Association, 2018, pp. 1353–1370.

[27] B. Bünz, S. Agrawal, M. Zamani, and D. Boneh, "Zether: Towards privacy in a smart contract world," in *Proceedings of the 2020 Financial Cryptography and Data Security (FC)*. Springer, 2020, pp. 423–443.

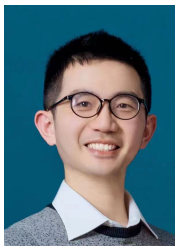
[28] Q. Ren, H. Liu, Y. Li, and H. Lei, "Cloak: A framework for development of confidential blockchain smart contracts," in *Proceedings of the 41st IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2021, pp. 1102–1105.
[29] T. Frassetto, P. Jauernig, D. Koisser *et al.*, "POSE: Practical off-chain smart contract execution," in *Proceedings of the 2023 Network and Distributed System Security Symposium (NDSS)*. ISOC, 2023, pp. 1–18.
[30] V. Buterin, "Ethereum: A secure decentralised generalised transaction ledger," https://ethereum.org/en/whitepaper/, 2013, Ethereum white paper, accessed: Apr., 2024.
[31] V. Costan and S. Devadas, "Intel SGX explained," https://eprint.iacr.org/2016/086, 2016, cryptology ePrint Archive, Paper 2016/086, accessed: Apr., 2023.
[32] S. Li, K. Xue, D. S. Wei *et al.*, "SecGrid: A secure and efficient SGX-enabled smart grid system with rich functionalities," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 1318–1330, 2019.
[33] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger (berlin version)," https://ethereum.github.io/yellowpaper/paper.pdf, 2022, Ethereum yellow paper, accessed: Apr., 2024.
[34] V. G. V Buterin, "Casper the friendly finality gadget," 2017, arXiv preprint. [Online]. Available: https://arxiv.org/abs/1710.09437
[35] "Ethereum statistics," https://ycharts.com/indicators/reports/ethereum_statistics, accessed: Apr., 2024.

Xinyi Luo received her B.S. degree in Information Security from School of the Gifted Young, University of Science and Technology of China (USTC), in 2020. She is currently working toward the Ph.D. degree in Information Security with the School of Cyber Science and Technology, USTC. Her research interests include Blockchain, Network security and Applied Cryptography.

Kaiping Xue (M'09-SM'15) received his bachelor's degree from the Department of Information Security, University of Science and Technology of China (USTC), in 2003 and received his doctor's degree from the Department of Electronic Engineering and Information Science (EEIS), USTC, in 2007. From May 2012 to May 2013, he was a postdoctoral researcher with the Department of Electrical and Computer Engineering, University of Florida. Currently, he is a Professor in the School of Cyber Science and Technology, USTC. He is also the director of Network and Information Center, USTC. His research interests include future Internet architecture design, transmission optimization and network security. He serves on the Editorial Board of several journals, including the IEEE Transactions on Dependable and Secure Computing (TDSC), the IEEE Transactions on Wireless Communications (TWC), and the IEEE Transactions on Network and Service Management (TNSM). He is an IET Fellow and an IEEE Senior Member.

Zhuo Xu received his B.S. degree in information security from School of Cyber Science and Technology, University of Science and Technology of China (USTC), in 2022. He is currently a graduate student in information security with the School of Cyber Science and Technology, USTC. His research interests include network security protocol design and analysis, applied cryptography, and blockchain.

Mingrui Ai received his B.S. degree in information security from the School of Cyber Science and Technology, University of Science and Technology of China (USTC), in 2020. He is currently working toward the Ph.D degree in information security with the School of Cyber Science and Technology, USTC. His research interests include network security analysis and blockchain.



Jianan Hong rececived the PhD degree from the Department of Electronic Engineering and Information Science (EEIS), USTC, in 2018. From 2018 to 2021, he was a Research Engineer with Huawei Shanghai Research Institute, Shanghai, China. He is currently an Assistant Researcher with The School of Cyber Science and Engineering, Shanghai Jiao Tong University, Shanghai, China. His research interests include secure cloud computing, privacy preserving authentication and lightweight zero-knowledge proof.

Xianchao Zhang received the Ph.D. degree in systems engineering from Beihang University, Beijing, China, in 2013. From 2013 to 2015, he was a Post-Doctoral Fellow with Peking University, Beijing. From 2018 to 2022, he was a Post-Doctoral Fellow with Southeast University, Nanjing, China. From 2015 to 2021, he was a Senior Engineer with the China Academy of Electronics and Information Technology, Beijing. He is currently a Professor with the Key Laboratory of Medical Electronics and Digital Health, Zhejiang, China. His research interests include artificial networks, wireless communications and and network security.

Qibin Sun (F'11) received the Ph.D. degree from the Department of Electronic Engineering and Information Science (EEIS), University of Science and Technology of China (USTC), in 1997. He is currently a professor in the School of Cyber Science and Technology, USTC. His research interests include multimedia security, network intelligence and security, and so on. He has published more than 120 papers in international journals and conferences. He is a fellow of IEEE.

Jun Lu received his bachelor's degree from southeast university in 1985 and his master's degree from the Department of Electronic Engineering and Information Science (EEIS), University of Science and Technology of China (USTC), in 1988. Currently, he is a professor in the School of Cyber Science and Technology and the Department of EEIS, USTC. He is also the president of Jiaxing University. His research interests include theoretical research and system development in the field of integrated electronic information systems, network and information security. He is an Academician of the Chinese Academy of Engineering (CAE).