

Bridging Programmability, Efficiency, and Bounded Trust: A Hybrid Privacy-Preserving Smart Contract Framework

Youheng Wang, *IEEE Student Member*, Rujia Li, *IEEE Member*, Zhaoyang Xie,
Kaikai Feng, Qingjie Chen, Yang Gao, Sisi Duan, *IEEE Senior Member*

Abstract—Privacy-preserving smart contracts (PPSCs) extend blockchain computation from transparent execution to confidential applications, enabling mutually distrustful parties to jointly compute contract logic on private inputs. Existing PPSC designs can be categorized into two main paradigms: trusted hardware-based systems and cryptographic systems. Trusted hardware-based systems provide general programmability and the performance is usually close to non-confidential computation, but the hardware has to be trusted. In contrast, cryptographic systems require much lower trust on the hardware but the performance is usually much lower.

In this paper, we propose a hybrid PPSC framework that combines trusted hardware with cryptographic techniques, achieving both general programmability and reduced reliance on trusted hardware. Specifically, the TEE executes the smart contracts, but needs to authenticate the computation. A proof of the encrypted computational results is sent on-chain, and the blockchain authenticates the computational and aggregates the computational results using cryptographic approaches such as homomorphic encryption. In this way, the confidential smart contract via TEE is both efficient and general programmable, without being trusted. Meanwhile, the on-chain cryptographic approach does not introduce high overhead as it only authenticates and aggregates the results. We formalize the system model and security goals, and prove the correctness using the Universal Composability framework. Our implementation and evaluation on Intel SGX as the trusted hardware and Solidity as the smart contract show that our approach achieves nearly no degradation on the performance compared to non-confidential computation.

1. Introduction

Privacy-preserving smart contracts (PPSCs) were introduced to extend blockchain functionality from transparent

on-chain computation to confidential, auditable, and verifiable applications [1], [2]. PPSCs enable mutually distrustful parties to jointly execute contract logic over private inputs while ensuring execution confidentiality, computational correctness, and public verifiability of the execution process. PPSCs have been widely used in various applications, including privacy-preserving auctions, confidential DeFi protocols [3], accountable decryption [4], credential systems [5], and privacy-preserving payments.[6].

Over the past few years, many PPSC schemes have been proposed, and we categorize existing works into two paradigms [7]: the cryptographic systems and trusted hardware-based systems. The cryptographic systems employ cryptographic approaches such as zero-knowledge proofs (ZKPs) and homomorphic encryption (HE) [8] to achieve public verifiability of the execution results without revealing confidential data. These approaches do not require strong trust on the underlying hardware, i.e., as long as the private keys are stored confidentially, the confidentiality of the data is achieved in a provably secure way. However, cryptographic confidential computing is known to be expensive and incur high computational cost. Meanwhile, trusted hardware-based approaches [9] leverage hardware such as Trusted Execution Environments (TEEs) to execute contracts inside isolated enclaves. If the hardware is fully trusted, confidentiality of the data is preserved while the execution results are also publicly verifiable. Two nice features about trusted hardware-based approach are general programmability and high performance [2], [10]. However, compared to cryptographic approaches, where the hardware for storing private keys needs to be trusted, trusted hardware-based approaches require the entire enclave to be trusted, which might be an overly strong assumption. Prior works have shown that trusted hardware-based smart contracts may suffer various vulnerabilities, making the data not confidential anymore [9], [11].

Thus, an interesting research question is:

Can we build a PPSC framework that enjoys the best of both worlds?

In this work, we answer this question affirmatively by proposing a hybrid PPSC framework called HyPSC, that preserves TEE-level general programmability and efficiency without fully trusting the hardware. HyPSC still uses TEE for executing the smart contract, same as trusted hardware-based approaches. Instead of fully trusting the enclave, our approach requires TEE to generate an attestation that au-

Youheng Wang and Kaikai Feng are with Shandong University, Qingdao, China. Emails: {youheng, 202321172}@mail.sdu.edu.cn. Rujia Li, Zhaoyang Xie, and Sisi Duan are with Tsinghua University, Beijing, China; Rujia Li and Sisi Duan are also with State Key Laboratory of Cryptography and Digital Economy Security; Sisi Duan is also with Zhongguancun Laboratory, Shandong Institute of Blockchains. Emails: {rujia, duansisi}@tsinghua.edu.cn, xiezhaoyang25@mails.tsinghua.edu.cn. Yang Gao, Qingjie Chen are with Digital Currency Institute of the People's Bank of China, Beijing, China. Emails: {gaoyang02, chenqingjie}@pbc.dci.cn. This work was supported in part by the National Key R&D Program of China (Grant No. 2023YFB2705000).

Corresponding authors: Rujia Li and Sisi Duan.

thenticates the data processing. Meanwhile, the computation results are handled on-chain using cryptographic primitives such as homomorphic encryption. Since the computation is handled by TEE, general programmability is achieved. Meanwhile, since cryptographic approaches are used to deal with the computational results, little overhead is introduced, and TEE does not have to be fully trusted. The workflow is shown in Figure 1.

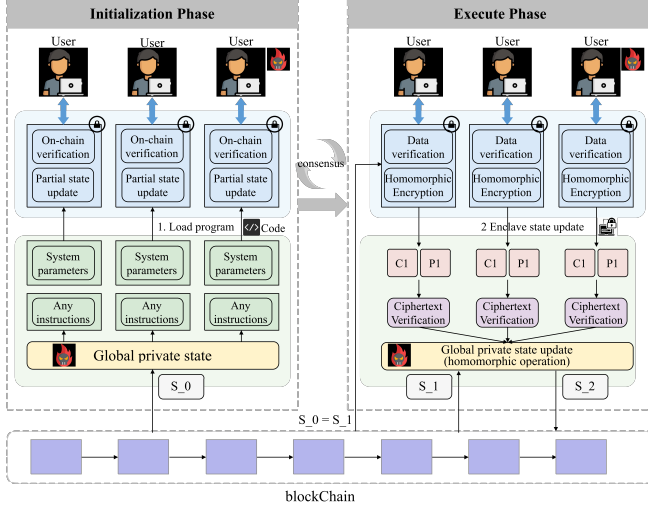


Figure 1: Overview of HyPSC, our hybrid privacy-preserving smart contract framework.

Technical challenges. Although the high-level idea is intuitive, formally specifying the security guarantees and proving correctness for our HyPSC framework is far from trivial. The framework involves multiple components such as private data processed inside TEEs, cryptographic proofs that attest to enclave behavior, on-chain verification embedded in smart contract logic, and secure computation that updates the encrypted global state. Direct analysis is challenging because the components operate under different trust assumptions. To address this challenge, we decompose the framework into a set of modular building blocks, inspired by prior TEE-based designs [2]. We then use the Universal Composability (UC) framework [12] to model HyPSC and prove the correctness of our approach.

To demonstrate the practicality of HyPSC, we design and implement two representative applications leveraging Intel SGX [13] and Ethereum’s smart contract [14]: privacy-preserving payment [6], [15] and blockchain-based e-voting [16], [17]. Our evaluation results show that our HyPSC framework is highly efficient, offering lower latency and reduced gas costs compared to known privacy-preserving smart contract frameworks [6], [18].

Our contributions. We make the following contributions.

- We propose a hybrid PPSC framework HyPSC that enjoys the benefits of trusted hardware-based PPSC and cryptographic PPSC: it has general programmability and high performance without the need to fully trust the underlying hardware (Section 3).

- We provide the first formalization of hybrid PPSC execution using the UC framework. We prove that our HyPSC framework achieves confidentiality of the data and verifiability of the correctness of execution results, even if TEE is compromised (§3.4).
- We provide two applications of our framework: privacy-preserving payment and blockchain-based e-voting (§4). We describe their abstract workflows and show how each is supported by instantiating application-specific components of the HyPSC.
- We implement a prototype of HyPSC using Intel SGX as the TEE and Solidity-based as the smart contract. Our evaluation results show that our protocol achieves significantly lower gas consumption and higher performance than known PPSC. The gas cost of the transfer operation in HyPSC is reduced by over 80%, and the transfer latency is reduced by over 90% compared to Zether [6] (§5).

2. System Model and Building Blocks

2.1. System Model

Smart contracts are executable programs with instances and states stored on the blockchain [19]. They allow transactions to be managed without a trusted third party. Privacy-preserving smart contracts [1], [6], [18] aim to protect the confidentiality of contract execution and the state stored on the blockchain. We distinguish two types of state:

- *Global state S* , which is maintained on-chain by the smart contract and shared among all users. In our framework, the global state is stored only in encrypted form.
- *Local state*, which consists of users’ private inputs and temporary computation results held by their clients or TEEs.

Our PPSC framework consists of the following roles:

- **Blockchain $\mathcal{B}$** maintains the distributed ledger and hosts smart contracts. It stores and updates global state in encrypted form, executes public contract logic, and verifies proofs from enclaves.
- **Users $\mathcal{U}$** possess private data and require privacy protection for contract state. We assume there are n users $\mathcal{U} = \{\mathcal{U}_i\}_{i \in [n]}$ in our protocol. They interact with $\mathcal{B}$ by submitting requests to smart contracts via local clients, while their private data remains protected.
- **Clients $\mathcal{P}$** runs applications on behalf of users. We assume there are n clients $\mathcal{P} = \{\mathcal{P}_i\}_{i \in [n]}$ in our protocol. Each client receives private inputs from users, invokes smart contracts on the blockchain, and executes the protocol logic.

The goals of PPSC. Our primary security objective is to protect the confidentiality of the encrypted global state, even in the presence of corrupted clients or enclaves. Formally, let $\Pi_{\text{PrivContract}}$ denote the real-world protocol (see Figure 5) and let $\mathcal{F}_{\text{PrivContract}}$ be an ideal functionality (see Figure 13). The ideal functionality maintains an internal state S , which is updated upon receiving task inputs task_input_i from clients.

For each invocation initiated by a client $\mathcal{P}_i$, $\mathcal{F}_{\text{PrivContract}}$ updates S according to the contract logic and outputs the prescribed public transcript. If $\mathcal{P}_i$ is corrupted, $\mathcal{F}_{\text{PrivContract}}$ leaks task_input_i to the adversary $\mathcal{A}$; If $\mathcal{P}_i$ is honest, it leaks nothing beyond the public outputs. In all cases, the joint view of $\mathcal{A}$ in the real-world execution is computationally indistinguishable from its view in the ideal execution with $\mathcal{F}_{\text{PrivContract}}$, except for information implied by corrupted inputs and the prescribed public outputs.

Trusted execution environment. TEE is an isolated processing environment that provides confidentiality and integrity guarantees for code and data [20]. Integrity ensures that enclave code executes as loaded and cannot be modified during the execution. Confidentiality ensures that sensitive data processed inside the TEE is protected from the host system, including the operating system and other applications. In the last decades, many TEE products have been proposed, such as Intel SGX [13], Intel TDX [21], AMD SEV-SNP [22], and Arm TrustZone [23].

Our new threat model for TEE. Side-channel attacks have posed a significant threat to the security of TEEs [24], [25]. Unlike previous works on trusted hardware [2] that assume both integrity and confidentiality for the TEE, our work assumes only integrity. We assume that an adversary $\mathcal{A}$ may launch side-channel attacks on the TEEs, potentially revealing the internal randomness of programs loaded in TEE. However, in this model, we assume $\mathcal{A}$ cannot cause the TEEs to produce incorrect outputs or steal the secret key used to generate the attestation. In practice, ensuring integrity requires a smaller trusted computing base than ensuring both integrity and confidentiality. Moreover, protecting the secret key for TEE authentication is easier than protecting all data within the TEEs, as the key is derived from secrets embedded in CPUs and is only accessible by well-designed and tested firmware from manufacturers.

2.2. Building Blocks

We use λ as the security parameter, and 1^λ for its unary representation. We sometimes omit the unary representation without raising ambiguity.

Signature scheme. A signature scheme S consists of:

- $S.\text{Gen}(1^\lambda) \rightarrow (pk, sk)$: on input the security parameter 1^λ , output a key pair consisting of a public verification key pk and a secret signing key sk .
- $S.\text{Sign}_{sk}(m) \rightarrow \sigma$: on input a message m and the signing key sk , output a signature σ on m .
- $S.\text{Verify}_{pk}(m, \sigma) \rightarrow \{0, 1\}$: on input a message m , a candidate signature σ , and the public key pk , output 1 if σ is a valid signature on m under pk , and 0 otherwise.

We assume that S satisfies existential unforgeability under chosen message attacks (EUF-CMA) [26].

Hash Function. A hash scheme H consists of:

- $H : \{0, 1\}^* \rightarrow \{0, 1\}^n$: a deterministic, efficiently computable function that maps an arbitrary-length input string to a fixed-length output of n bits.

We require that H satisfies standard security properties, including pre-image resistance, second pre-image resistance, and collision resistance [27].

Homomorphic encryption. An Homomorphic Encryption scheme HE has four algorithms:

- $\text{HE.Gen}(1^\lambda) \rightarrow (pk, sk)$: on input 1^λ , output a public/secret key pair.
- $\text{HE.Enc}_{pk}(m) \rightarrow c$: encrypt message m with public key pk to obtain c .
- $\text{HE.Dec}_{sk}(c) \rightarrow m$: decrypt c with secret key sk to recover m (or $\perp$).

HE schemes exhibit two basic homomorphic properties. A scheme is *additively homomorphic* if there exists a ciphertext operation $\oplus$ such that, for all m_1, m_2 , $\text{HE.Enc}_{pk}(m_1) \oplus \text{HE.Enc}_{pk}(m_2) = \text{HE.Enc}_{pk}(m_1 + m_2)$. A scheme is *multiplicatively homomorphic* if there exists a ciphertext operation $\odot$ such that, for all m_1, m_2 , $\text{HE.Enc}_{pk}(m_1) \odot \text{HE.Enc}_{pk}(m_2) = \text{HE.Enc}_{pk}(m_1 \cdot m_2)$. The concrete operator on ciphertexts is scheme-dependent. For brevity, we denote these homomorphic operations explicitly as $\text{HE.Add}(c_1, c_2) \triangleq c_1 \oplus c_2$, $\text{HE.Mul}(c_1, c_2) \triangleq c_1 \odot c_2$. Thus, HE.Add performs encrypted addition, and HE.Mul performs encrypted multiplication on ciphertexts.

We assume that HE is IND-CPA secure [28]. HE should be resistant to chosen plaintext attacks. That means an adversary cannot distinguish between the encryption of two different messages.

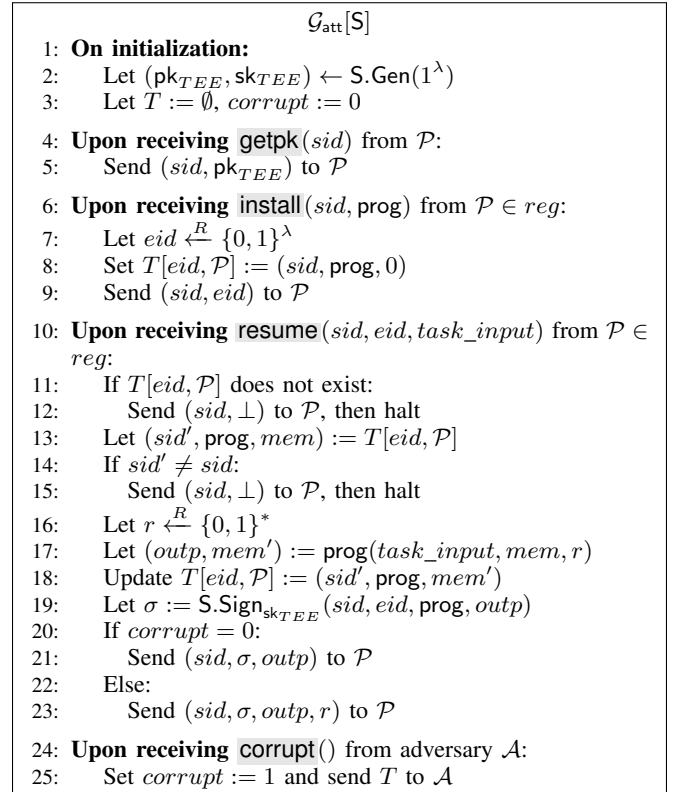


Figure 2: Ideal functionality of trusted execution environments

TEE abstraction. We model trusted execution environments

using an ideal functionality $\mathcal{G}_{\text{att}}$, inspired by prior abstractions [29], [30], [31], as shown in Figure 2. $\mathcal{P}$ equipped with TEEs (denoted by a set reg), may install programs prog into TEEs via `install` calls. Once installed, a program prog can be executed on given inputs task_input via `resume` calls, using the session identifier sid and the enclave identifier eid . The outputs of prog are signed using the private key sk_{TEE} of the functionality. Other parties can verify that an output outp was produced by a program installed in $\mathcal{G}_{\text{att}}$ using the corresponding public key pk_{TEE} . The ideal functionality always executes installed programs prog according to their specified code and produces verifiable outputs. Unless explicitly corrupted, its internal state cannot be modified.

Blockchain. We model the blockchain as an append-only ledger functionality $\mathcal{F}_{\text{Ledger}}[\text{succ}]$ (see Figure 3), inspired by the abstraction in Ekiden [2]. The functionality maintains an internal storage Storage indexed by identifiers id . Parties can query the stored value associated with an identifier by invoking `read`, and can request state updates by invoking `write` with input task_input . Upon a write request, the functionality checks a successor relation succ on the current stored value. The parameter succ defines the validity of new items. A new item can only be appended to the storage if the evaluation of succ outputs 1. If the check succeeds, the storage is updated and a receipt is returned. Otherwise, the request is rejected. The functionality enforces the successor relation on all updates and ensures that only valid state transitions are recorded.

$\mathcal{F}_{\text{Ledger}}[\text{succ}]$ – Append-only Blockchain Ledger

- 1: **Parameter:** successor relationship $\text{succ} : \{0, 1\}^* \times \{0, 1\}^* \rightarrow \{0, 1\}$
- 2: **Init:** $\text{Storage} \leftarrow \emptyset$
- 3: **Upon receiving** `read`(id):
- 4: **if** $\text{id} \in \text{Storage}$ **then**
- 5: Output $\text{Storage}[\text{id}]$
- 6: **else**
- 7: Output $\perp$
- 8: **end if**
- 9: **Upon receiving** `write`($\text{id}, \text{task_input}$) :
- 10: Let $\text{val} \leftarrow \text{Storage}[\text{id}]$
- 11: **if** $\text{succ}(\text{val}, \text{task_input}) = 1$ **then**
- 12: $\text{Storage}[\text{id}] \leftarrow \text{val} \parallel (\text{task_input}, \mathcal{P})$
- 13: Output (`receipt`, id)
- 14: **else**
- 15: Output (`reject`, id)
- 16: **end if**

Figure 3: Ideal functionality $\mathcal{F}_{\text{Ledger}}[\text{succ}]$.

2.3. Universal Composability Framework

The UC framework, introduced by Canetti in 2001 [12], provides a security notion ensuring that cryptographic protocols remain secure even when arbitrarily composed with others. In the UC framework, security is defined with respect to an ideal functionality, which represents a trusted and idealized specification of the protocol.

Security is formalized via an indistinguishability game between a real-world execution and an ideal-world execution. An interactive environment $\mathcal{Z}$ provides inputs to the parties and observes their outputs. In the real world, $\mathcal{Z}$ interacts with the protocol parties and an adversary $\mathcal{A}$, whereas in the ideal world, $\mathcal{Z}$ interacts with the ideal functionality and a simulator $\mathcal{S}$ that emulates the adversary’s behavior. A protocol is said to UC-realize an ideal functionality if, for every PPT adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that no PPT environment $\mathcal{Z}$ can distinguish between the real and ideal executions with non-negligible probability.

Our setting naturally fits the UC paradigm. Our framework involves both off-chain and on-chain components: private data is processed and verified off-chain, while encrypted computation and state updates are performed on-chain. These components operate in different environments and exhibit different trust and security properties. As a result, a unified and composable abstraction is required to model their interactions and to reason about correctness and privacy in a principled way.

3. System Overview

3.1. Our Framework

Our framework is shown in Figure 4. As mentioned in the introduction, our primary objective is to develop a hybrid privacy-preserving smart contract system that combines the advantages of TEE and cryptographic primitives.

In our framework, each client includes a trusted execution environment `enclave`¹. Our protocol uses trusted execution environments for high-performance privacy operations. This avoids heavy cryptographic overhead. Instead of one enclave accessing all private data, our approach limits each enclave to processing a subset of private data. Each enclave generates an attestation to authenticate the computation. The enclave’s results are sent to the blockchain for execution. Our protocol also uses homomorphic operations to manipulate encrypted data on-chain. This approach allows smart contracts to update the global state without exposing sensitive information. It ensures the confidentiality of the on-chain state.

By combining enclave-backed computation with homomorphic ciphertext processing, our design cleanly separates trust assumptions: TEEs provide efficient off-chain computation. The data is then handled on-chain using cryptographic primitives such as homomorphic encryption. Our system does not fully rely on TEE confidentiality for security. Compromising a single enclave may reveal a subset of private data, but it does not expose whole private data or affect system security.

3.2. The Workflow

Our protocol consists of two phases: the initialization phase and the execution phase.

1. Here, we use “enclave” to refer to the secure execution space within the TEE.

In the initialization phase, a client sets up a local trusted enclave and loads a program `prog`. The program `prog` defines the contract logic to be executed inside the enclave. The program provides an initialization function `Init`, which initializes the enclave's local state and generates an attestation proof. The proof binds the enclave's identity to a public key pk_{TEE} . The client submits the attestation proof σ_{att} to the blockchain. The contract verifies the attestation and records the trust relationship between the client and the enclave. This phase establishes a trusted execution context that can later produce verifiable computation results.

In the execution phase, the enclave executes `prog` on private inputs maintained in its local state. The program `prog` provides an execution function `execute`, which processes private data locally (e.g., performing homomorphic encryption `HE.Enc` or decryption `HE.Dec`) and generates an execution proof σ_{exec} with respect to the registered code and current state. The enclave outputs an encrypted task description `task` together with σ_{exec} . The client submits $(task, \sigma_{exec})$ to the blockchain. The contract verifies the execution proof and updates the on-chain state from S_{old} to S_{new} by performing computation directly over ciphertexts. State transitions are accepted only if they are backed by valid enclave executions.

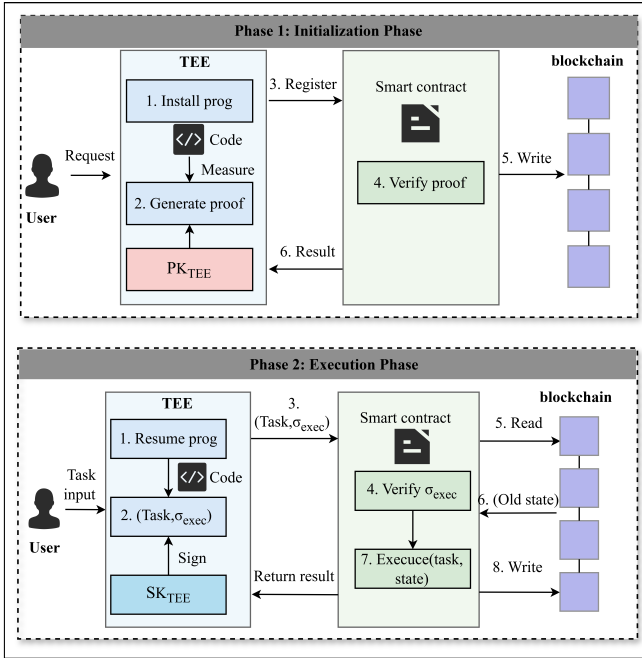


Figure 4: System architecture of HyPSC.

3.3. Security Definition and Analysis

We define the security of our protocol, with the formal proof provided in Appendix A. Our main security goal is to ensure that the encrypted global state remains confidential, even if some enclaves are compromised. We use $\Pi_{PrivContract}$ to denote the real-world protocol and use $\mathcal{F}_{PrivContract}$ to de-

note the ideal functionality. Formally, we define the security property as follows:

Theorem 1 (Security of HyPSC). Assume that the attestation mechanism provided by $\mathcal{G}_{att}$ and the signature scheme S are existentially unforgeable under chosen message attacks (EU-CMA). We further assume that the encryption schemes in our protocol are indistinguishable under chosen plaintext attacks (IND-CPA). Then $\Pi_{PrivContract}$ securely realizes $\mathcal{F}_{PrivContract}$ in the $(\mathcal{G}_{att}, \mathcal{F}_{Ledger})$ -hybrid model, for static adversaries $\mathcal{A}$.

3.4. Formal Specification of the Protocol

We show our protocol $\Pi_{PrivContract}$ formally in Figure 5. The protocol involves a set of clients $\{\mathcal{P}_i\}$, each equipped with a TEE, and blockchain system.

In the initialization phase, the client $\mathcal{P}_i$ installs the enclave program `prog` by invoking $\mathcal{G}_{att}.install$ (described in Figure 2). After installation, the client retrieves the enclave's signing public key pk_{TEE} via $\mathcal{G}_{att}.getpk$ (line 7 in Figure 5). The client then calls $\mathcal{G}_{att}.resume(sid, eid, (Init, pk_{TEE}))$ to complete initialization. During this call, `prog` generates the attestation proof σ_{att} (line 8 in Figure 5). The client submits `register`(`cid`, `addr`, pk_{TEE} , σ_{att}) to the blockchain for registration (line 9 in Figure 5). Here, `addr` represents the client's on-chain identity.

Upon receiving a valid registration request, the blockchain calls `Contract.register` to verify the attestation proof and register the enclave (line 22 in Figure 5). After the verification is successful, the blockchain then executes $\mathcal{F}_{Ledger}.write((state, addr), \perp)$ to store the initial account state and executes $\mathcal{F}_{Ledger}.write((trust, addr), pk_{TEE})$ to mark pk_{TEE} as trusted (lines 25,26 in Figure 5).

In the execution phase, `prog` processes data inside the enclave according to the specific application. The contract verifies the processed data and performs an encrypted computation. Finally, the results are recorded on the blockchain.

Depending on the concrete application, the client $\mathcal{P}_i$ prepares the input `task_inputi` and retrieves pk_{in}^{cid} from the blockchain. The client invokes $\mathcal{G}_{att}.resume(sid, eid, (execute, pk_{in}^{cid}, task_input_i))$ to execute `prog`. The enclave program `prog` verifies the input's validity, such as whether the balance exceeds the transaction amount in the privacy-preserving payment scenario. The program generates an encrypted task description `task` using an encryption scheme and generates a proof σ_{exec} (line 9 in Figure 5). The proof σ_{exec} is used to verify the correctness of the computation.

The blockchain retrieves the old state $S_{old} = \mathcal{F}_{Ledger}.read(task_i)$ (line 29 in Figure 5), which is the state before the latest update. The blockchain then invokes `Contract.execute`(`taski`, σ_{exec} , S_{old}) to verify the enclave attestation proof. After the verification is successful, the contract performs computation directly encrypted data. The computation can be used by cryptographic primitives such as homomorphic encryption and derives a state update operation `op` (line 31 in Figure 5).

$\Pi_{\text{PrivContract}}(\lambda, S, \{\mathcal{P}_i\}_{i \in [n]})$	
Clients $\mathcal{P}_i$:	Blockchain $\mathcal{B}$:
1: On receive (create, Contract) from environment $\mathcal{Z}$:	16: On input (create, Contract):
2: // assert cid initialized on blockchain	17: $cid \leftarrow H(\text{Contract})$
3: $cid \leftarrow \mathcal{B}.\text{create}(\text{Contract})$	18: $\mathcal{F}_{\text{Ledger}}.\text{write}(cid, \text{Contract})$
4: output (receipt, cid)	19: return (receipt, cid)
5: On receive (Init, cid , sid , prog) from environment $\mathcal{Z}$:	20: On input (register, $addr$, pk_{TEE} , σ_{att}):
6: $(sid, eid) \leftarrow \mathcal{G}_{att}.\text{install}(sid, prog)$	21: $\text{Contract} \leftarrow \mathcal{F}_{\text{Ledger}}.\text{read}(cid)$
7: $pk_{TEE} \leftarrow \mathcal{G}_{att}.\text{getpk}(sid)$	22: $res \leftarrow \text{Contract}.\text{register}(addr, pk_{TEE}, \sigma_{att})$
8: $(sid, \sigma_{att}) \leftarrow \mathcal{G}_{att}.\text{resume}(sid, eid, (Init, pk_{TEE}))$	23: if $res \neq \text{ok}$ then return (err, res)
9: $(result, addr) = \mathcal{B}.\text{register}(cid, addr, pk_{TEE}, \sigma_{att})$	24: end if
10: output (result, $addr$)	25: $\mathcal{F}_{\text{Ledger}}.\text{write}((state, addr), \perp)$
11: On receive (execute, cid , sid , eid , $task_input_i$) from environment $\mathcal{Z}$:	26: $\mathcal{F}_{\text{Ledger}}.\text{write}((trust, addr), pk_{TEE})$
12: //get pk_{in}^{cid} from blockchain	27: return (ok, $addr$)
13: $(sid, \sigma_{exec}, task_i) \leftarrow \mathcal{G}_{att}.\text{resume}(sid, eid, (execute, pk_{in}^{cid}, task_input_i))$	28: On input (execute, cid , $task_i$, σ_{exec}):
14: $(result, taskid) = \mathcal{B}.\text{execute}(cid, task_i, \sigma_{exec})$	29: $\text{Contract} \leftarrow \mathcal{F}_{\text{Ledger}}.\text{read}(cid)$
15: output (result, $taskid$)	30: $S_{old} \leftarrow \mathcal{F}_{\text{Ledger}}.\text{read}(task_i)$
	31: $(res, op, taskid) \leftarrow \text{Contract}.\text{execute}(task_i, \sigma_{exec}, S_{old})$
	32: if $res \neq \text{ok}$ then return (err, res)
	33: end if
	34: $\mathcal{F}_{\text{Ledger}}.\text{write}(task_i, op)$
	35: return (ok, $taskid$)

Figure 5: Formal specification of protocol $\Pi_{\text{PrivContract}}$.

The blockchain applies the update by executing $\mathcal{F}_{\text{Ledger}}.\text{write}(task_i, op)$, and then returns (res, $taskid$) to the client (lines 34,35 in Figure 5). Finally, the client updates its local private state inside the enclave.

4. Two Concrete Instantiations

In this section, we present two concrete real-world applications of HyPSC: *privacy-preserving on-chain payment* [6], [15] and *blockchain-based e-voting* [16], [17]. Before delving into the technical details, we first clarify the choice of cryptographic building blocks. We would like to emphasize that our applications are used to validate the effectiveness and generality of our HyPSC framework, instead of providing more efficient privacy-preserving on-chain payment or blockchain-based e-voting. Indeed, as mentioned previously, our goal is to enjoy the general programmability of TEE, but remove the assumption on fully trusted hardware via cryptographic algorithms.

4.1. Privacy-Preserving Payment

Privacy-preserving on-chain payment aims to enable the transfer of cryptocurrency without revealing sensitive information such as balances and transaction amounts. Many privacy-preserving smart contract frameworks [6], [18] have been developed for on-chain payments, including cryptography-based approaches [6], [18] and trusted hardware-based approaches [2], [32]. These applications suffer from the same challenges as mentioned previously. For instance, compromising one enclave may expose the privacy of many users. Recently, Jean-Louis et al.[11] proposed attacks targeting vulnerabilities such as access pattern

leakage and state consistency issues in TEE-based smart contract platforms.

4.1.1. The Workflow. We now sketch the workflow of the privacy-preserving payment built upon our protocol. The formal protocol is described in §4.1.2.

The system maintains two separate but linked views of a user's balance:

- bal_{on} : an encrypted balance stored and maintained on the blockchain. In our case, it is updated through homomorphic encryption.
- bal_{off} : a local, plaintext balance maintained inside the enclave. It reflects spendable funds and is used for fast decision-making.

In this protocol, we assume that each client is equipped with a TEE and can communicate securely with the blockchain. We denote a client as $\mathcal{P}_i$ and the blockchain as $\mathcal{B}$. This section provides the concrete realization of the protocol specified in Section 3.4. The protocol consists of three phases. The Initialization phase in Section 3.4 is instantiated as the Initialization phase in this section. The Execution phase in Section 3.4 is instantiated as two phases, Withdrawal and Transfer.

Initialization phase. In this phase, the client $\mathcal{P}_i$ installs the enclave program and registers a trusted enclave on chain.

a) *Enclave creation:* Client $\mathcal{P}_i$ creates a secure enclave. Inside the enclave, it generates two key pairs: one for signing (pk_{TEE}, sk_{TEE}) and one for homomorphic encryption (pk_{HE}, sk_{HE}). The private keys are kept inside the enclave.

b) *Local state initialization:* The enclave loads the code, which serves as the basis for subsequent proof verification and transaction validation. Meanwhile, the offchain available

balance bal_{off} and nonce are set to 0, where nonce is used to bind subsequent messages and prevent replay attacks.

c) *Trust establishment*: After the initialization, the enclave performs attestation to prove its integrity to the blockchain. The enclave sends an attestation proof σ_{att} , which confirms that the TEE is running the correct code. The blockchain verifies the proof accordingly. When the proof is accepted, the enclave's public key pk_{TEE} is stored on-chain as a trusted identity. It will be used to verify future transactions from the client $\mathcal{P}_i$.

Withdrawal phase. In this phase, client $\mathcal{P}_i$ withdraws the balance from the on-chain to the TEE.

a) *On-chain state update*: $\mathcal{P}_i$ sends a withdrawal request to the smart contract. The contract retrieves the encrypted balance Ce . This balance is encrypted using the client's homomorphic encryption key pk_{HE} . The contract computes a new balance Ce' , which represents a zero balance. This confirms that the on-chain balance has been consumed. Once the transaction is confirmed on-chain, the tuple (Ce, Ce') is sent to the client.

b) *Enclave state update*: The client receives (Ce, Ce') through a secure communication channel. The enclave program uses sk_{HE} to decrypt Ce' and confirms that its plaintext value is 0. This confirms that the encrypted balance has been correctly computed on-chain. The program computes the difference $t = \text{HE.Dec}_{\text{sk}_{\text{HE}}}(\text{Ce}) - \text{HE.Dec}_{\text{sk}_{\text{HE}}}(\text{Ce}')$.

Transfer phase. This phase enables a user-to-user transfer.

a) *Enclave state check*: The enclave checks whether $\text{bal}_{\text{off}} \geq v_t$, where v_t is the requested transfer amount. If not, the request is rejected. If the condition holds, the enclave updates the off-chain balance as $\text{bal}_{\text{off}}' = \text{bal}_{\text{off}} - v_t$, and marks the state as pending to prevent double spending. The state is finalized only after confirmation on-chain. The enclave encrypts the transfer amount as $\Delta C = \text{HE.Enc}_{\text{pk}_{\text{HE}}}(v_t)$ and signs the request using sk_{TEE} .

b) *On-chain state update*: The signed transaction, along with ΔC , is submitted to the smart contract. The contract verifies the signature and checks that the sender enclave is authenticated and trusted. If all checks pass, it updates the recipient's encrypted balance as $\text{Ce}'_{\text{recv}} = \text{Ce}_{\text{recv}} \oplus \Delta C$.

c) *Enclave state update*: Once the transaction is confirmed, the enclave finalizes the off-chain balance by clearing the pending state.

4.1.2. Formal Specification of the Protocol. Our protocol $\Pi_{\text{PrivPay}}(\lambda, \text{HE}, S, \{\mathcal{P}_i\}_{i \in [n]})$ in the UC model is defined in Figure 6. Π_{PrivPay} relies on the TEE program prog_{pay} in Figure 7 and the smart contract in Figure 8.

Initialization phase. All other steps follow the initialization procedure described above in §3.4. This instantiation extends the enclave logic to support homomorphic key generation and payment-specific state initialization. During this phase, the enclave program prog_{pay} generates a homomorphic encryption key pair $(\text{pk}_{\text{HE}}, \text{sk}_{\text{HE}})$ (line 2 in Figure 7). The program initializes the client's off-chain balance as bal_{off} and samples a fresh nonce nonce (lines 3,4 in Figure 7). Then, the client registers the account $\text{addr} \leftarrow H(\text{pk}_{\text{HE}})$ on the blockchain.

Upon receiving a register request, the blockchain invokes $\text{Contract}_{\text{pay}}.\text{register}(\text{pk}_{\text{HE}}, \text{pk}_{\text{TEE}}, \sigma_{\text{att}})$. All verification steps follow the procedure described in §3.4. If registration succeeds, the blockchain updates the ledger by initializing the user's encrypted balance and recording the trusted enclave (lines 29-31 in Figure 6).

Withdrawal phase. To withdraw funds, the client invokes $\mathcal{G}_{\text{att}}.\text{resume}(\text{sid}, \text{eid}, \text{initiate withdraw})$ to obtain a withdrawal transaction and its attestation proof. The enclave prepares a withdrawal task task_i by deriving the addr , updating nonce , and generating an attestation proof σ_{wd} for the task (lines 8-10 in Figure 7). The client then submits a withdrawal request $(\text{cid}, \text{task}_i, \sigma_{\text{wd}})$ to the blockchain.

The blockchain retrieves the encrypted balance bal_{on} associated with the account addr and forwards task_i , the attestation proof σ_{wd} , and bal_{on} to the contract (lines 34,35 in Figure 6). If verification succeeds, the contract computes a new encrypted balance Ce' (line 11 in Figure 8). The blockchain updates the ledger accordingly and returns Ce' to the client (lines 38-41 in Figure 6). Finally, the client invokes $\mathcal{G}_{\text{att}}.\text{resume}(\text{sid}, \text{eid}, \text{withdraw}, \text{Ce}')$, and the enclave updates the local off-chain balance accordingly (lines 12-19 in Figure 7).

Transfer phase. The client invokes $\mathcal{G}_{\text{att}}.\text{resume}(\text{sid}, \text{eid}, \text{transfer}, (v_t, \text{addr}_{\text{send}}, \text{addr}_{\text{recv}}, \text{pk}_{\text{HE}}^{\text{recv}}))$, where v_t is the transfer amount, $\text{addr}_{\text{send}}$ is the sender's address, $\text{addr}_{\text{recv}}$ is the recipient's address, and $\text{pk}_{\text{HE}}^{\text{recv}}$ is the recipient's homomorphic encryption public key. The enclave program prog_{pay} checks whether the local balance is sufficient. If the check passes, prog_{pay} encrypts the transfer amount ΔC for the recipient, generates a signed transfer task_i . The enclave then updates its balance to a pending state $\text{bal}_{\text{pending}} = \text{bal}_{\text{off}} - v_t$, awaiting confirmation of the transfer transaction success (lines 21-27 in Figure 7). The client sends the transfer request $(\text{task}_i, \sigma_{\text{tr}})$ to the blockchain.

The blockchain retrieves the recipient's encrypted balance Ce and forwards $(\text{task}_i, \sigma_{\text{tr}}, \text{Ce})$ to the contract. The contract verifies the signature and checks that nonce is fresh. If the verification succeeds, the contract computes $\text{Ce}' \leftarrow \text{HE.Add}(\text{Ce}, \text{task}_i.\Delta C)$ (lines 14-21 in Figure 8). The blockchain updates the ledger by the contract execution result (line 47 in Figure 6). Finally, the blockchain returns $(\text{confirm}, \text{taskid})$ to the enclave. Upon confirmation, the enclave finalizes the transfer by converting the $\text{bal}_{\text{pending}}$ to the bal_{off} (line 31 in Figure 7).

4.2. Blockchain-based E-Voting

Blockchain-based e-voting systems use the blockchain to enhance the security, transparency, and integrity of the voting process. These systems aim to prevent fraud and manipulation, ensure voter anonymity, and increase trust in elections [33]. However, they face a conflict between protecting voter identity and voting content while maintaining auditability and transparency [34], [35]. Additionally, it is essential to mitigate threats and prevent the fraudulent use or unauthorized disclosure of sensitive voter data [36], [37]. Scalability is important in such systems since the number of

$\Pi_{\text{PrivPay}}(\lambda, H, HE, S, \{\mathcal{P}_i\}_{i \in [n]})$

Clients $\mathcal{P}_i$:

```

1: On receive (create,  $\text{Contract}_{\text{pay}}$ ) from environment  $\mathcal{Z}$ :
2: // assert  $cid$  initialized on blockchain
3:  $cid \leftarrow \text{create}(\text{Contract}_{\text{pay}})$ 
4: output (receipt,  $cid$ )

5: On receive (Init,  $cid, sid, \text{prog}$ ) from environment  $\mathcal{Z}$ 
6:  $(sid, eid) \leftarrow \mathcal{G}_{\text{att}}.\text{install}(sid, \text{prog})$ 
7:  $\text{pk}_{TEE} \leftarrow \mathcal{G}_{\text{att}}.\text{getpk}(sid)$ 
8:  $(sid, \sigma_{\text{att}}, \text{pk}_{HE}) \leftarrow \mathcal{G}_{\text{att}}.\text{resume}(sid, eid, (\text{pk}_{TEE}, \text{Init}))$ 
9:  $(\text{result}, \text{addr}) = \mathcal{B}.\text{register}(cid, \text{pk}_{HE}, \text{pk}_{TEE}, \sigma_{\text{att}})$ 
10: output (result,  $\text{addr}$ )

11: On receive (withdraw,  $cid, sid, eid$ ) from environment  $\mathcal{Z}$ :
12:  $(sid, \sigma_{\text{wd}}, \text{task}_i) \leftarrow \mathcal{G}_{\text{att}}.\text{resume}(sid, eid, \text{initiate withdraw})$ 
13:  $(\text{Ce}, \text{Ce}') = \mathcal{B}.\text{withdraw}(cid, \text{task}_i, \sigma_{\text{wd}})$ 
14:  $\mathcal{G}_{\text{att}}.\text{resume}(sid, eid, \text{withdraw}, (\text{Ce}, \text{Ce}'))$ 
15: output (Ce,  $\text{Ce}'$ )

16: On receive (transfer,  $sid, eid, cid, v_t, \text{addr}_{\text{send}}, \text{addr}_{\text{recv}}, \text{pk}_{HE}^{\text{recv}}$ ) from environment  $\mathcal{Z}$ :
17:  $(sid, \sigma_{\text{tr}}, \text{task}_i) \leftarrow \mathcal{G}_{\text{att}}.\text{resume}(sid, eid, (\text{transfer}, v_t, \text{addr}_{\text{send}}, \text{addr}_{\text{recv}}, \text{pk}_{HE}^{\text{recv}}))$ 
18:  $(\text{confirm}, \text{taskid}) = \mathcal{B}.\text{transfer}(cid, \text{task}_i, \sigma_{\text{tr}})$ 
19:  $\mathcal{G}_{\text{att}}.\text{resume}(sid, eid, (\text{confirm}, \text{taskid}))$ 
20: output (confirm,  $\text{taskid}$ )

```

Blockchain $\mathcal{B}$:

```

21: On input (create,  $\text{Contract}_{\text{pay}}$ ):
22:  $cid \leftarrow H(\text{Contract}_{\text{pay}})$ 
23:  $\mathcal{F}_{\text{Ledger}}.\text{write}(cid, \text{Contract}_{\text{pay}})$ 
24: return (receipt,  $cid$ )

25: On input (register,  $\text{pk}_{HE}, \text{pk}_{TEE}, \sigma_{\text{att}}$ ):
26:  $\text{res} \leftarrow \text{Contract}_{\text{pay}}.\text{register}(\text{pk}_{HE}, \text{pk}_{TEE}, \sigma_{\text{att}})$ 
27: if  $\text{res} \neq \text{ok}$  then return reject
28: end if
29:  $\text{addr} \leftarrow H(\text{pk}_{HE})$ 
30:  $\mathcal{F}_{\text{Ledger}}.\text{write}((\text{balance}, \text{addr}), \perp)$ 
31:  $\mathcal{F}_{\text{Ledger}}.\text{write}((\text{trust}, \text{addr}), \text{pk}_{TEE})$ 
32: return (ok,  $\text{addr}$ )

33: On input (withdraw,  $cid, \text{task}_i, \sigma_{\text{wd}}$ ):
34:  $\text{bal}_{\text{on}} \leftarrow \mathcal{F}_{\text{Ledger}}.\text{read}(\text{task}_i.\text{addr})$ 
35:  $(\text{res}, \text{op}, \text{Ce}') \leftarrow \text{Contract}_{\text{pay}}.\text{withdraw}(\text{task}_i, \sigma_{\text{wd}}, \text{bal}_{\text{on}})$ 
36: if  $\text{res} \neq \text{ok}$  then return reject
37: end if
38:  $\text{op} = (\text{write}, \text{task}_i.\text{addr}, \text{Ce}')$ 
39:  $\mathcal{F}_{\text{Ledger}}.\text{write}((\text{balance}, \text{task}_i.\text{addr}), \text{Ce}')$ 
40:  $\text{Ce} = \text{bal}_{\text{on}}$ 
41: return (Ce,  $\text{Ce}'$ )

42: On input (transfer,  $cid, \text{task}_i, \sigma_{\text{tr}}$ ):
43:  $\text{Ce} \leftarrow \mathcal{F}_{\text{Ledger}}.\text{read}(\text{task}_i.\text{addr}_{\text{recv}})$ 
44:  $(\text{res}, \text{op}) \leftarrow \text{Contract}_{\text{pay}}.\text{transfer}(\text{task}_i, \sigma_{\text{tr}}, \text{Ce})$ 
45: if  $\text{res} \neq \text{ok}$  then return reject
46: end if
47:  $\mathcal{F}_{\text{Ledger}}.\text{write}(\text{op})$ 
48: return (confirm,  $\text{taskid}$ )

```

Figure 6: Formal specification of protocol Π_{PrivPay} , an application of the $\Pi_{\text{PrivContract}}$ in privacy-preserving payment scenario.

voters might be large [38], [39]. Our solution is a good fit for such an application. In our concrete instantiation, we batch the votes and use homomorphic encryption to update the encrypted voting state. This approach resolves the conflict between privacy and auditability, reduces the risk of vote leakage, and enhances the system's scalability.

4.2.1. Workflow. We now sketch the blockchain-based e-voting workflow using our protocol. The formal protocol is described in Section 4.2.2.

The chain keeps a registry indexed by a hash of each voter's public key; it stores a nullifier and a nonce to prevent double voting. The enclave keeps a small local state to track per-voter freshness and whether a vote was already cast.

The workflow has two phases: initialization and execution.

Initialization phase. This phase sets up trust relationship and parameters.

a) Enclave creation: Client $\mathcal{P}_i$ creates a secure enclave. Inside the enclave, it generates a signing key pair $(\text{pk}_{TEE}, \text{sk}_{TEE})$. The private key never leaves the enclave.

b) Local state initialization: The enclave loads $\text{prog}_{\text{vote}}$ and initializes local state, including a *voted* map to record voted users and a *nonce* to prevent replay attacks.

c) Trust establishment: The enclave performs remote attestation and obtains a proof σ_{att} . The blockchain verifies the proof and records pk_{TEE} on chain as a trusted identity for subsequent message verification.

Execution phase. This phase casts a vote on-chain while keeping the choice secret.

a) Enclave state checks: The user signs the candidate they are voting for with their private key outside the enclave. The enclave program then verifies the user's signature. If the verification is successful, the program computes the hash of the user's public key and checks that the request is within the valid time window. Next, the program verifies that the user has not previously voted. After these checks, the enclave prepares an attested spend message that contains the hash and the homomorphically encrypted candidate vector $(c_1, \dots, c_k)$. The actual voting choice never leaves the enclave.

b) On-chain state update: The client sends the attested message to the contract. The contract verifies that the enclave is trusted and that the user's public key has not been used. If all checks pass, it marks the user as voted and updates the on-chain ballot by adding the homomorphically encrypted candidate vector $(t_1, \dots, t_k)$.

c) Ballot decryption: When the election period ends, the

```

Client program  $\text{prog}_{\text{pay}}$  inside  $\mathcal{G}_{\text{att}}$ 
1: On input (Init) :
2:  $(\text{pk}_{HE}, \text{sk}_{HE}) \leftarrow \text{HE.Gen}(1^\lambda)$ 
3:  $\text{nonce} \leftarrow \text{Random}(\{0, 1\}^\lambda)$ 
4: Initialize  $\text{bal}_{\text{off}} \leftarrow 0$ 
5: return  $\text{pk}_{HE}$ 

6: On input (initiate withdraw):
7:  $\text{addr} \leftarrow \text{H}(\text{pk}_{HE})$ 
8:  $\text{nonce} \leftarrow \text{nonce} + 1$ 
9:  $\text{task} \leftarrow (\text{addr}, \text{nonce}, \text{pk}_{HE})$ 
10: return  $\text{task}$ 

11: On input (withdraw,  $\text{Ce}, \text{Ce}'$ ):
12:  $x' \leftarrow \text{HE.Dec}_{\text{sk}_{HE}}(\text{Ce}')$ 
13: if  $x' \neq 0$  then
14:   return reject
15: end if
16:  $t \leftarrow \text{HE.Dec}_{\text{sk}_{HE}}(\text{Ce}) - \text{HE.Dec}_{\text{sk}_{HE}}(\text{Ce}')$ 
17:  $\text{bal}_{\text{off}} \leftarrow \text{bal}_{\text{off}} + t$ 
18: return ok

19: On input (transfer,  $v_t, \text{addr}_{\text{send}}, \text{addr}_{\text{recv}}, \text{pk}_{HE}^{\text{recv}}$ ):
20: if  $\text{bal}_{\text{off}} < v_t$  then
21:   return Reject
22: end if
23:  $\Delta C \leftarrow \text{HE.Enc}_{\text{pk}_{HE}^{\text{recv}}}(v_t)$ 
24:  $\text{nonce} \leftarrow \text{nonce} + 1$ 
25:  $\text{task} \leftarrow (\text{addr}_{\text{send}}, \text{addr}_{\text{recv}}, v_t, \Delta C, \text{nonce})$ 
26:  $\text{bal}_{\text{pending}} \leftarrow \text{bal}_{\text{off}} - v_t$ 
27: return  $\text{task}$ 

28: On input (confirm,  $\text{taskid}$ ):
29: // transaction confirmed on-chain
30:  $\text{bal}_{\text{off}} \leftarrow \text{bal}_{\text{pending}}$ 
31: return ok

```

Figure 7: Client enclave program prog_{pay}

authority retrieves the encrypted ballot from the ledger. It decrypts each homomorphic tally using its secret key to obtain the plaintext vote counts $(v_1, \dots, v_k)$ for all candidates.

4.2.2. Formal Specification of the Protocol. Our protocol $\Pi_{\text{PrivVote}}(\lambda, S, \{\mathcal{P}_i\}_{i \in n})$ in the $(\mathcal{G}_{\text{att}}, \mathcal{F}_{\text{Blockchain}})$ hybrid model is defined in Figure 9 to Fig. 11. Π_{PrivVote} relies on the TEE program $\text{prog}_{\text{vote}}$ in Figure 10 and on the smart contract $\text{Contract}_{\text{vote}}$ in Figure 11.

Initialization phase. All other steps follow the initialization procedure described above in §3.4. This instantiation extends the enclave logic to support blockchain-based e-voting.

During the initialization, the enclave program $\text{prog}_{\text{vote}}$ initializes a local map `voted` to record whether a $\mathcal{U}_i$ public key $\text{pk}_{\mathcal{U}_i}$ has already voted (line 23 in Figure 10). Then, the client registers the account pk_{TEE} on the blockchain.

Upon receiving a `register` request, the blockchain invokes $\text{Contract}_{\text{vote}}.\text{register}(\text{pk}_{TEE}, \sigma_{\text{att}})$. All verification steps follow the procedure described in §3.4. If registration succeeds, the blockchain records the trusted enclave identity associated with pk_{TEE} (line 23 in Figure 9).

Execution phase. User $\mathcal{U}_i$ produces a signature $\sigma_{\text{user}} =$

```

Contract $_{\text{pay}}$ 
1: On input (register,  $\text{pk}_{HE}, \text{pk}_{TEE}, \sigma_{\text{att}}$ ):
2: if  $\text{S.Verify}_{\text{pk}_{TEE}}(\sigma_{\text{att}}) = 0$  then return reject
3: end if
4:  $\text{Trusted}[\text{pk}_{HE}] \leftarrow \text{pk}_{TEE}$ 
5: return ok

6: On input (withdraw,  $\text{task}, \sigma_{\text{wd}}, \text{Ce}$ ):
7:  $\text{pk}_{TEE} \leftarrow \text{Trusted}[\text{task.addr}]$ 
8:  $\text{pk}_{HE} = \text{task.pk}_{HE}$ 
9: if  $\text{S.Verify}_{\text{pk}_{TEE}}(\text{task}, \sigma_{\text{wd}}) = 0$  then return reject
10: end if
11:  $\text{Ce}' \leftarrow \text{HE.Enc}_{\text{pk}_{HE}}(0)$ 
12: return (ok, (write,  $\text{task.addr}, \text{Ce}'$ ),  $\text{Ce}'$ )

13: On input (transfer,  $\text{task}, \sigma_{\text{tr}}, \text{Ce}$ ):
14:  $\text{pk}_{TEE} \leftarrow \text{Trusted}[\text{task.addr}_{\text{send}}]$ 
15: if  $\text{S.Verify}_{\text{pk}_{TEE}}(\text{task}, \sigma_{\text{tr}}) = 0$  then return reject
16: end if
17: if  $\text{tx.nonce} \in \text{NonceList}[\text{task.addr}_{\text{send}}]$  then return reject
18: end if
19:  $\text{Ce}' \leftarrow \text{HE.Add}(\text{Ce}, \text{task}.\Delta C)$ 
20:  $\text{NonceList}[\text{task.addr}_{\text{send}}].\text{append}(\text{task.nonce})$ 
21: return (ok, (write,  $\text{task.addr}_{\text{recv}}, \text{Ce}'$ ))

```

Figure 8: Blockchain smart contract $\text{Contract}_{\text{pay}}$

$\text{S.Sign}_{\text{sk}_{\mathcal{U}_i}}(\text{eid} \parallel j)$ to prove ownership of $\text{pk}_{\mathcal{U}_i}$, where j is the candidate that $\mathcal{U}$ voted for (line 12 in Figure 9). The client $\mathcal{P}_i$ then invokes the voting program $\text{prog}_{\text{vote}}$ to prepare a voting task inside the enclave. Inside the enclave, $\text{prog}_{\text{vote}}$ verifies the user's identity, checks that the user has not voted before and constructs an encrypted ballot vector under the election public key $\text{pk}_{HE}^{\text{cid}}$ (lines 7-21 in Figure 10). If all checks pass, the enclave outputs a voting task task_i together with an attestation proof σ_{exec} . The client submits $(\text{cid}, \text{task}_i, \sigma_{\text{exec}})$ to the blockchain.

Upon receiving a `vote` request, the blockchain calls $\text{Contract}_{\text{vote}}.\text{vote}(\text{task}_i, \text{pk}_{TEE}, \sigma_{\text{tee}})$. In addition to the verification steps described in §3.4, The contract further checks whether the user has already voted and rejects the vote if a duplicate submission is detected (lines 8-15 in Figure 11). If all checks pass, the contract updates the encrypted tally (lines 15-17 in Figure 11). The blockchain updates the ledger by the contract execution result.

Upon a `reveal` request, the system uses the secret key $\text{sk}_{HE}^{\text{cid}}$ to decrypt the stored ballots and publish the election outcome (lines 31-38 in Figure 9).

5. Implementation and Evaluation

5.1. Implementation

We build the HyPSC in Solidity and C++. We use Ethereum as the blockchain. We use Intel SGX[13] as the TEE instance. We use SGX SDK (version 2.26.100)² to manage SGX Enclaves and SGXSSL³ for in-enclave

2. <https://github.com/intel/linux-sgx>

3. <https://github.com/intel/intel-sgx-ssl>

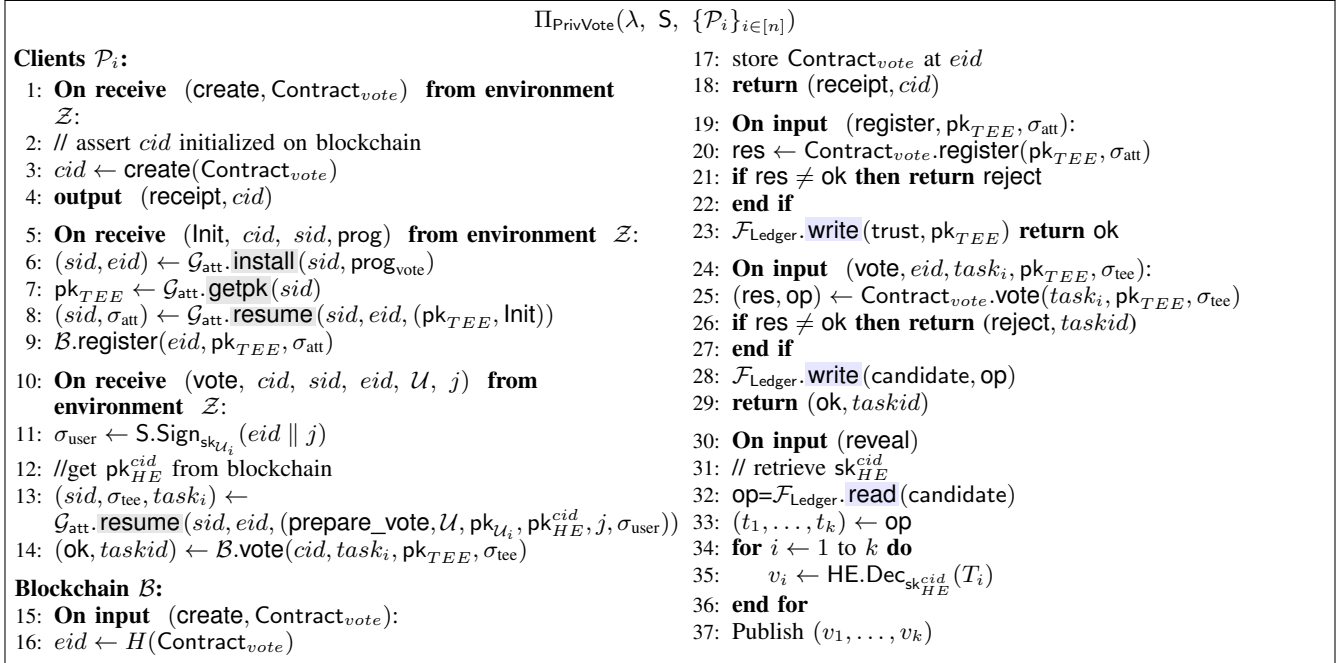


Figure 9: Formal specification of protocol $\Pi_{PrivVote}$, an application of the $\Pi_{PrivContract}$ in Block-based e-voting.

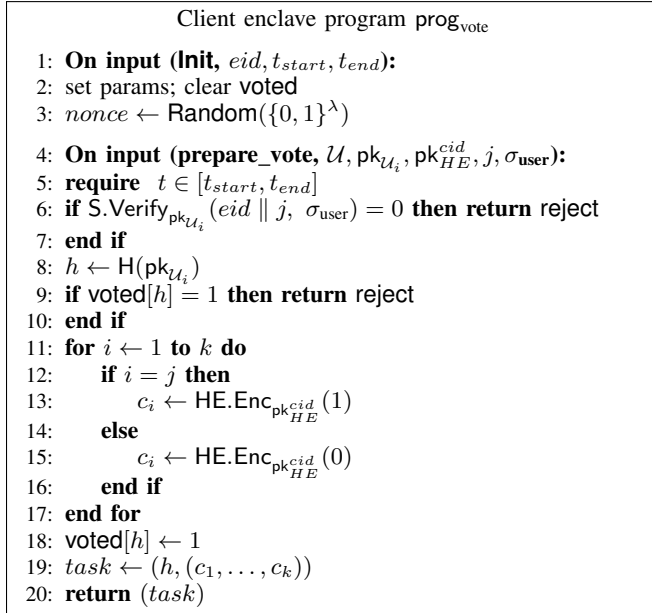


Figure 10: Client enclave program prog_{vote}

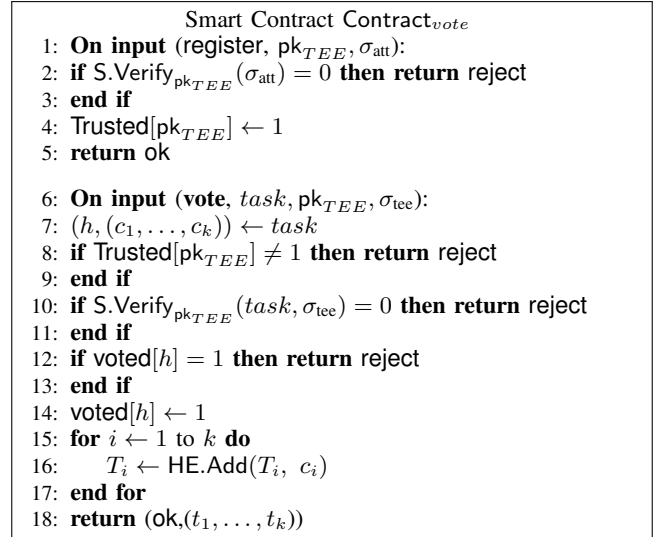


Figure 11: Block-based e-voting Contract. $h = H(pk_{U_i})$ is recorded; a user key can cast at most one vote, regardless of the TEE used.

cryptographic operations.

To enhance the efficiency of cryptographic primitives, we employ a hybrid cryptographic scheme. The homomorphic encryption scheme is ElGamal over the BN254 curve, matching the Ethereum precompiles [40]. The digital signature scheme is ECDSA over secp256k1 [41], with Keccak-256 [42] as the hash function.

The enclave code is written in C++ for Intel SGX. The

on-chain smart contract is written in Solidity and compiled with solc 0.8.28⁴. The protocol in privacy-preserving payment scenario consists of 1,837 lines of code, while the blockchain-based e-voting scenario consists of 832 lines of code. We have made our implementation publicly available at <https://github.com/fu4ure/privacy-contract>.

4. <https://docs.soliditylang.org/en/v0.8.30/>

5.2. Evaluation

We evaluate the performance and cost of HyPSC, covering enclave cryptographic operations, on-chain gas consumption, and end-to-end latency in two representative application scenarios: privacy-preserving payment and blockchain-based e-voting. Additionally, we present a theoretical performance comparison among HyPSC, Zether [6] (a privacy-preserving payment approach), and Secret Network⁵ (a TEE-based smart contract approach). We compare the transfer methods of Zether, the SNIP-20⁶ on the Secret Network, and our protocol. HyPSC reduces transfer latency by over 99.83% compared to Zether and 99.39% compared to the Secret Network.

Experimental settings. We implement our scheme and evaluate it on an Intel® Xeon® Platinum 8369B @ 2.70 GHz (4 vCPUs: 2 cores × 2 threads) with 7 GB RAM, running Ubuntu 22.04.4 LTS (Linux 5.15.0-117-generic). HyPSC and Zether is deployed to a local Hardhat testnetwork⁷, which is used for testing Ethereum smart contracts. Unless otherwise specified, every node proposes a block every 2 seconds.

Theoretical performance comparison. We compare the time complexities of the Zether[6], the Secret Network, and our HyPSC. The comparison focuses on the computationally expensive operations, including encryption and proof generation, on both the user side and the blockchain side. Throughout this discussion, we use the following notation: p denotes the ciphertext modulus used in cryptographic operations, N represents the number of constraints in the zero-knowledge proof circuit, and $|m|$ is the size of the message involved in the signing or verification process. The comparison shows that Zether has higher complexity due to zero-knowledge proofs. Secret Network avoids this overhead but requires full trust in enclave confidentiality. HyPSC uses ElGamal encryption and ECDSA signature generation with lower complexity.

In Zether, the user-side operations include ElGamal encryption[43], with a time complexity of $O(\log p)$. Bulletproof[44] generation has a time complexity of $O(N)$. Schnorr[45] signature generation takes $O(\log p + |m|)$. On the blockchain side, Bulletproof verification has a time complexity of $O(N)$, and Schnorr signature verification takes $O(\log p + |m|)$. Updates to the account and pending transfers, based on ElGamal homomorphic computation, each have a time complexity of $O(\log p)$.

For Secret Network, privacy-preserving execution relies entirely on Trusted Execution Environments. On the user side, clients perform symmetric encryption and message authentication with time complexity $O(|m|)$. Validators execute contract logic inside enclaves. On the blockchain side, verification includes enclave attestation and signature checking with complexity $O(\log p + |m|)$. Secret Network avoids zero-knowledge proofs and incurs no $O(N)$ overhead, but

requires full trust in enclave confidentiality, since global balances and transfer amounts are processed in plaintext inside TEEs.

In HyPSC, on the user side, ElGamal encryption is performed with a time complexity of $O(\log p)$. ECDSA[46] signature generation on the secp256k1 curve takes $O(\log p)$, and Keccak256 hashing has a time complexity of $O(|m|)$. On the blockchain side, ECDSA signature verification takes $O(\log p + |m|)$, and account updates involve ElGamal homomorphic computations with a complexity of $O(\log p)$.

Latency of cryptographic operations in the enclave. Figure 12a reports the latency of enclave side cryptographic operations as the batch size increases from 1 kB to 4096 kB. For secp256k1_sign, latency stays in 400–420 μ s over the range. For bn254_elgamal_enc, latency remains 1.12 ms across all sizes. For bn254_elgamal_dec, latency ranges from 415 to 430 μ s and shows no growth with input size. For keccak256_1KB, latency is 111–115 μ s across all settings. The figure shows that signing and decryption curves are flat, indicating that latency is stable as the batch size increases.

Throughput. We vary the number of transactions submits to the protocol, denoted as send rate. Figure 12b presents the throughput for our protocol at different send rates. After the send rate reaches 600, throughput stabilizes around 120 tx/s. Further increases in the send rate do not significantly boost throughput.

Additionally, we vary the intervals of block proposals (denoted as block time). Figure 12c shows that as block time increases, throughput degrades. At a block time of 2, TPS is 131.63 tx/s, but it decreases to 48.60 tx/s at a block time of 14.

Method	Gas	Request Size (B)
deployment	3,274,836	
register	167,724	10,028
transfer	97,415	544
withdraw	154,783	758

TABLE 1: Gas usage for payment contract methods.

Measurement of privacy-preserving payment contract Table 1 reports the gas cost of the main contract methods together and the corresponding request sizes. Deployment requires about 3.27M gas, and occurs once when the contract is published. Among the interactive methods, register consumes 167k gas with a request size of 1028 B, withdraw costs 154k gas with 758 B input, and transfer costs 97k gas with 544 B input.

The register method and withdraw consume more gas than transfer because they perform an additional encryption step to establish the encrypted state on chain. transfer performs fewer state writes and thus costs least. Request sizes match payload content: register request includes attestation and public keys; withdraw request includes a nonce and a signature; transfer request includes ciphertexts, a signature, and recipient keys. These results indicate that initialization heavy operations are gas intensive, while routine transfers are cheaper.

5. <https://github.com/scrtlabs/SecretNetwork>

6. <https://github.com/scrtlabs/snip20-reference-impl>

7. <https://hardhat.org/>

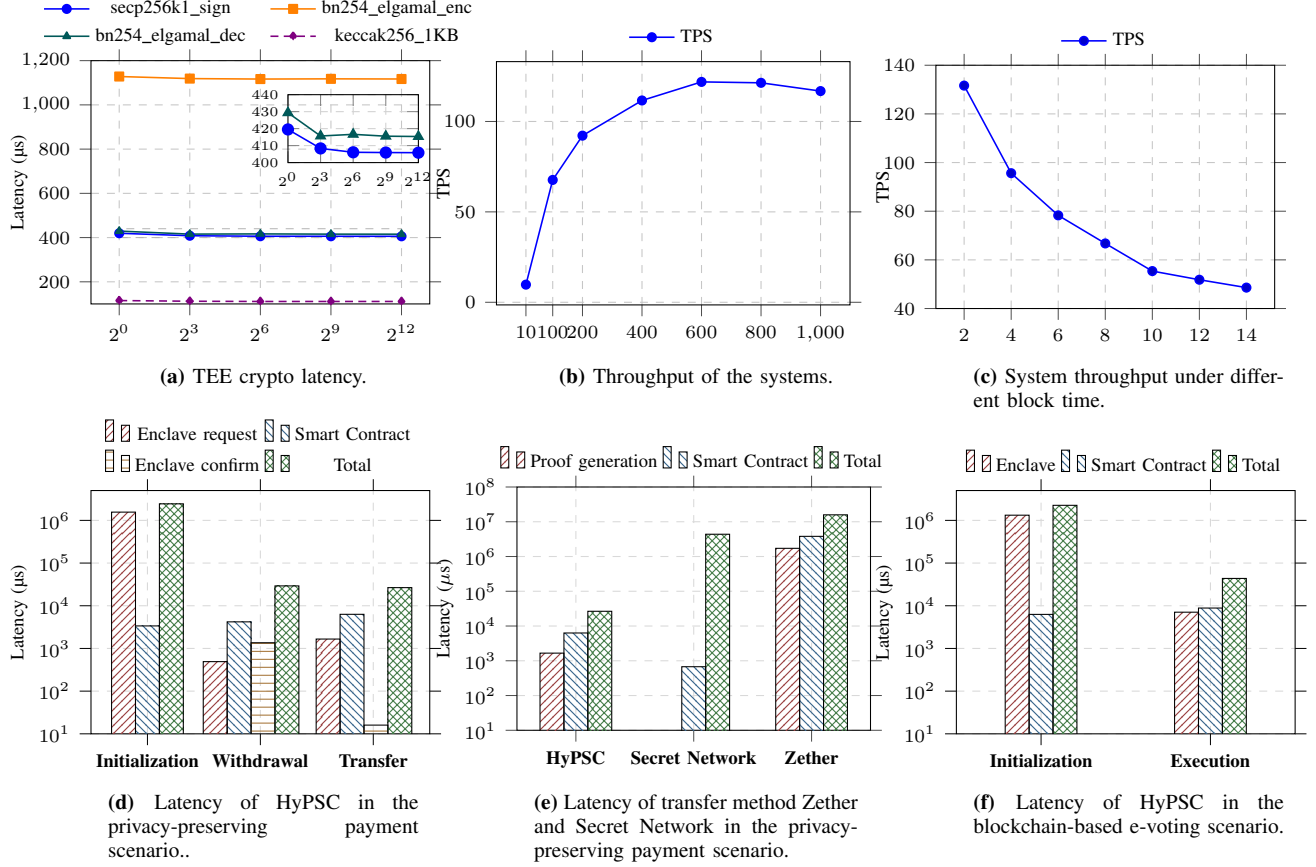


Figure 12: Latency comparison of HyPSC and Zether across privacy-preserving payment and e-voting scenarios.

In comparison to Zether [6], which requires 7,188k gas and 1472 bytes of request size for the **transfer**, our approach shows a lower gas cost for **transfer** (97k gas and 544 bytes of request size). This is primarily because our smart contract replaces the verification of zero-knowledge proofs with signature verification.

Time cost of the privacy-preserving payment protocol. Figure 12d reports the end-to-end latency of each phase in HyPSC. Initialization phase has the highest latency (2,439,490 μ s). This is mainly due to enclave setup and key generation. The withdrawal phase takes 29,185 μ s, and the transfer phase takes 26,568 μ s. In these two phases, the cost is mainly associated with smart contract execution.

Figure 12e compares the transfer latency of HyPSC, Zether, and Secret Network. Since Secret Network is designed to work with its blockchain, we did not use Ethereum for testing. Zether has higher transfer latency due to zero-knowledge proof generation and on-chain verification. We also evaluated the transfer phase of the Secret Network SNIP-20 token. The transfer phase has a smart contract execution time of 674,926 μ s, while the end-to-end latency reaches 4,736,000 μ s. HyPSC completes the transfer phase within 26,568 μ s.

HyPSC incurs latency primarily from enclave-side operations during initialization and from smart contract execu-

tion during state transitions. In contrast, Zether introduces additional latency due to zero-knowledge proof generation and on-chain verification, which dominate the execution time in the transfer phase. The Secret Network contract executes efficiently in TEE, but its overall latency depends on factors such as the consensus mechanism.

Method	Gas	Request Size (B)
deployment	1,617,701	
register	49,819	9,879
vote	547,919	1052

TABLE 2: Gas usage for vote contract methods

Measurement of the blockchain-based e-voting contract. Our testing assumes five candidates in the election. Table 2 shows the gas cost of the main voting methods and their request sizes. Deployment costs about 1.61M gas. The **register** method uses 49.8k gas with a 9.88 kB request, while **vote** uses 547k gas with a 1 kB input. The larger **register** request includes the TEE attestation and a public key. The **vote** method involves signature verification and event emission, which makes its gas cost slightly higher.

Time cost of the blockchain-based e-voting protocol. Figure 12f shows the latency per phase. The initialization phase takes 2,252,795 μ s. This is mainly due to enclave setup,

Category	Approach	Confidentiality	Programmability	Performance		
				Implementation	Generate Proof	Verify Proof
Crypto-based	Kosba et al. [1]	Provable security	need customized circuits	BCTV14[47]	$O(n \log n)$	$O(1)$
	Banerjee et al. [48]	Provable security	need customized circuits	sigma protocols[49]	$O(n)$	$O(n)$
	Bünz et al. [6]	Provable security	need customized circuits	Σ -Bulletproof [6]	$O(n)$	$O(n)$
	Bowe et al. [50]	Provable security	need customized circuits, only support UTXO model	GM17[51]	$O(n \log n)$	$O(1)$
	Xiong et al. [52]	Provable security	need customized circuits	Plonk[53]	$O(n \log n)$	$O(1)$
	Steffen et al. [54]	Provable security	only Zasm programs are supported.	GM17[51]	$O(n \log n)$	$O(1)$
	Solomon et al. [18]	Provable security	need customized circuits	DPLS19 [55] and BulletProofs	$O(n)$	$O(n)$
	Jeong et al. [56]	Provable security	need customized circuits	Groth16 [57]	$O(n \log n)$	$O(1)$
TEE-based	Jiang et al. [58]	Provable security	need customized circuits	Groth16 [57]	$O(n \log n)$	$O(1)$
	Yuan et al. [32]	Trusted hardware assumption	General-purpose programmability	Intel SGX [13]	$O(1)$	$O(1)$
	Bowman et al. [59]	Trusted hardware assumption	General-purpose programmability	Intel SGX [13]	$O(1)$	$O(1)$
	Das et al. [60]	Trusted hardware assumption	General-purpose programmability	Intel SGX [13]	$O(1)$	$O(1)$
	Cheng et al. [2]	Trusted hardware assumption	General-purpose programmability	Intel SGX [13]	$O(1)$	$O(1)$
	Yan et al. [61]	Trusted hardware assumption	General-purpose programmability	Intel SGX [13]	$O(1)$	$O(1)$
	Müller et al. [62]	Trusted hardware assumption	General-purpose programmability	Arm TrustZone [23]	$O(1)$	$O(1)$
	Jian et al. [63]	Trusted hardware assumption	General-purpose programmability	Arm TrustZone [23]	$O(1)$	$O(1)$
	Our Work	Bounded trust hardware	General-purpose programmability	Intel SGX [13]	$O(1)$	$O(1)$

TABLE 3: Comparison of Confidentiality, Programmability, and Performance across Approaches. In the performance section, "Implementation" refers to the time complexity of implementing the system, "Generate Proof" represents the time complexity for proof generation, and "Verify Proof" represents the time complexity for proof verification. n denotes the original domain size.

proof generation, and verification. The execution phase takes 43,659 μ s, with the contract's delay caused by homomorphic encryption calculations for processing encrypted votes.

6. Related Work

Cryptography-based PPSC. Kosba et al. [1] proposed a framework for privacy-preserving smart contracts that uses a semi-trusted manager to process private data. The manager generates zero-knowledge proofs to convince the blockchain of correct execution without revealing private data. Banerjee et al. [48] replaced the centralized manager with a secure multi-party computation protocol. Bünz et al. [6] proposed an account-based confidential payment scheme compatible with Ethereum, using ElGamal encryption [43] and Σ -Bullet zero-knowledge proofs [44] to hide transfer amounts and balances. Bowe et al. [50] presented a general-purpose framework for decentralized private computation that used a record-based model and a universal zero-knowledge executor to hide both data and program logic. Xiong et al. [52] replaced per-application trusted setups with a universal one and optimized verification circuits for scalability. Steffen et al. [54] introduced Zapper, which provides a high-level programming model and unified Zasm execution to improve programmability and modularity while preserving zero-knowledge verification. However, Zapper supports only Zasm programs and does not support features such as pointer arithmetic, self-modifying code, or recursion. Solomon et al. [18] proposed a fully homomorphic encryption-based system for privacy-preserving smart contracts, where users submit encrypted inputs with zero-knowledge proofs and blockchain nodes compute on ciphertexts. The system supports lightweight clients and general encrypted computation but is limited by current FHE efficiency. Jeong et al. [56] proposed Azeroth, which combines privacy and auditabil-

ity by using a dual-account system, zkp, and a threshold encryption-based auditing mechanism. Jiang et al. [58] used a two-layer commitment structure and zkp to balance privacy and regulation, with practical implementation on an account-based blockchain.

TEE-based PPSC. TEEs provide strong isolation and high efficiency for confidential computation. They have been widely used in privacy-preserving smart contracts [2], [32], [62], [59], [60], [61], [63] to ensure confidentiality and improve performance. Yuan et al. [32] proposed ShadowEth, which stored contract binaries and states in off-chain TEE-DS, with an on-chain bounty contract for verification and rewards. Bowman et al. [59] presented Private Data Objects, executing contracts off chain in stateless TEEs to ensure data privacy. Das et al. [60] introduced FASTKITTEN, enabling complex Bitcoin contracts in TEEs with time-locked transactions and deposit penalties. Cheng et al. [2] designed Ekiden, a TEE-blockchain hybrid where TEEs ran contracts off chain and the chain verified encrypted states. Yan et al. [61] presented CONFIDE, a production-ready confidential contract engine deployed on AntChain, combining secure transmission, encryption, and key management protocols. Müller et al. [62] and Jian et al. [63] developed privacy-preserving frameworks based on Arm TrustZone [23]. Müller et al. designed TZ4Fabric, which integrated TrustZone with Hyperledger Fabric for Internet of Things (IoT) applications. Jian et al. proposed TSC-VEE, which enabled Solidity contracts to execute inside TrustZone.

In Table 3, we summarize the above approaches and compare them with our work in terms of confidentiality, programmability, and performance.

Our work enjoys the benefits of both cryptography-based PPSC and TEE-based PPSC, i.e., it operates under a bounded trust hardware assumption, while still achieves general programmability and high performance.

7. Conclusion

In this work, we propose a hybrid privacy-preserving smart contract framework that integrates TEE-based execution with cryptographic algorithms, achieving high performance and general programmability from TEE and low trust on trusted hardware from cryptographic algorithms. We provide a UC-based security proof for our approach, and show two applications in privacy-preserving payment and blockchain-based e-voting. A prototype implementation on Intel SGX and Solidity validates the effectiveness of our approach.

References

- [1] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, “Hawk: The blockchain model of cryptography and privacy-preserving smart contracts,” in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 839–858.
- [2] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song, “Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts,” in *2019 IEEE European symposium on security and privacy (EuroS&P)*. IEEE, 2019, pp. 185–200.
- [3] N. Tyagi, A. Arun, C. Freitag, R. Wahby, J. Bonneau, and D. Mazières, “Riggs: Decentralized sealed-bid auctions,” in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1227–1241.
- [4] R. Li, Q. Wang, F. Liu, Q. Wang, and D. Galindo, “An accountable decryption system based on privacy-preserving smart contracts,” in *International Conference on Information Security*. Springer, 2020, pp. 372–390.
- [5] R. Li, D. Galindo, and Q. Wang, “Auditable credential anonymity revocation based on privacy-preserving smart contracts,” in *International Workshop on Data Privacy Management*. Springer, 2019, pp. 355–371.
- [6] B. Bünz, S. Agrawal, M. Zamani, and D. Boneh, “Zether: Towards privacy in a smart contract world,” in *International Conference on Financial Cryptography and Data Security*. Springer, 2020, pp. 423–443.
- [7] H. Qi, M. Xu, D. Yu, and X. Cheng, “Sok: Privacy-preserving smart contract,” *High-Confidence Computing*, vol. 4, no. 1, p. 100183, 2024.
- [8] D. Aronoff, A. Bhat, P. Chatzigiannis, M. Minaei, S. Raghuraman, R. M. Townsend, and N. X.-Y. Zhang, “Sok: Fully-homomorphic encryption in smart contracts,” *Cryptology ePrint Archive*, 2025.
- [9] R. Li, Q. Wang, Q. Wang, D. Galindo, and M. Ryan, “Sok: Tee-assisted confidential smart contract,” *Proceedings on Privacy Enhancing Technologies*, vol. 3, pp. 711–731, 2022.
- [10] M. Tran, L. Luu, M. S. Kang, I. Bentov, and P. Saxena, “Obscuro: A bitcoin mixer using trusted execution environments,” in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 692–701.
- [11] N. Jean-Louis, Y. Li, Y. Ji, H. Malvai, T. Yurek, S. Bellemare, and A. Miller, “Sgxonerated: Finding (and partially fixing) privacy flaws in tee-based smart contract platforms without breaking the tee,” *Proceedings on Privacy Enhancing Technologies*, 2024.
- [12] R. Canetti, “Universally composable security: A new paradigm for cryptographic protocols,” in *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 2001, pp. 136–145.
- [13] V. Costan and S. Devadas, “Intel sgx explained,” *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 86, 2016. [Online]. Available: <https://api.semanticscholar.org/CorpusID:28642809>
- [14] V. Buterin *et al.*, “Ethereum white paper,” *GitHub repository*, vol. 1, no. 22–23, pp. 5–7, 2013.
- [15] K. Gai, Y. Guo, J. Yu, W. Chan, L. Zhu, Y. Zhang, and W. Meng, “Cape: Commitment-based privacy-preserving payment channel scheme in blockchain,” *IEEE Transactions on Dependable and Secure Computing*, 2025.
- [16] P. McCorry, S. F. Shahandashti, and F. Hao, “A smart contract for boardroom voting with maximum voter privacy,” in *International conference on financial cryptography and data security*. Springer, 2017, pp. 357–375.
- [17] Y. Li, W. Susilo, G. Yang, Y. Yu, D. Liu, X. Du, and M. Guizani, “A blockchain-based self-tallying voting protocol in decentralized iot,” *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 1, pp. 119–130, 2020.
- [18] R. Solomon, R. Weber, and G. Almashaqbeh, “smartfhe: Privacy-preserving smart contracts from fully homomorphic encryption,” in *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2023, pp. 309–331.
- [19] V. Y. Kemmoe, W. Stone, J. Kim, D. Kim, and J. Son, “Recent advances in smart contracts: A technical overview and state of the art,” *IEEE Access*, vol. 8, pp. 117 782–117 801, 2020.
- [20] M. Sabt, M. Achemlal, and A. Bouabdallah, “Trusted execution environment: What it is, and what it is not,” in *2015 IEEE Trust-com/BigDataSE/Ispaa*, vol. 1. IEEE, 2015, pp. 57–64.
- [21] “Intel® trust domain extensions (intel® tdx) module : Base architecture specification,” Intel Corporation, Tech. Rep., 2023. [Online]. Available: <https://cdrdv2.intel.com/v1/dl/getContent/853287?fileName=intel-tdx-module-base-spec-348549006.change-marks.pdf>
- [22] A. Sev-Snp, “Strengthening vm isolation with integrity protection and more,” *White Paper; January*, vol. 53, no. 2020, pp. 1450–1465, 2020.
- [23] “Arm security technology: Building a secure system using trustzone technology,” Arm Limited, Tech. Rep., 2009. [Online]. Available: <https://developer.arm.com/documentation/PRD29-GENC-009492/latest>
- [24] A. Muñoz, R. Ríos, R. Román, and J. López, “A survey on the (in) security of trusted execution environments,” *Computers & Security*, vol. 129, p. 103180, 2023.
- [25] S. Van Schaik, A. Seto, T. Yurek, A. Batori, B. AlBassam, D. Genkin, A. Miller, E. Ronen, Y. Yarom, and C. Garman, “Sok: Sgx fail: How stuff gets exposed,” in *2024 IEEE symposium on security and privacy (SP)*. IEEE, 2024, pp. 4143–4162.
- [26] S. Goldwasser, S. Micali, and R. L. Rivest, “A digital signature scheme secure against adaptive chosen-message attacks,” *SIAM Journal on computing*, vol. 17, no. 2, pp. 281–308, 1988.
- [27] J. Katz and Y. Lindell, *Introduction to modern cryptography: principles and protocols*. Chapman and hall/CRC, 2007.
- [28] S. Goldwasser and S. Micali, “Probabilistic encryption & how to play mental poker keeping secret all partial information,” in *Providing sound foundations for cryptography: on the work of Shafi Goldwasser and Silvio Micali*, 2019, pp. 173–201.
- [29] R. Pass, E. Shi, and F. Tramer, “Formal abstractions for attested execution secure processors,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2017, pp. 260–289.
- [30] F. Tramer, F. Zhang, H. Lin, J.-P. Hubaux, A. Juels, and E. Shi, “Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge,” in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2017, pp. 19–34.
- [31] X. Hu, R. Li, Y. Liu, and Q. Wang, “Towards efficient and practical multi-party computation under inconsistent trust in tees,” in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2025, pp. 2885–2902.
- [32] R. Yuan, Y.-B. Xia, H.-B. Chen, B.-Y. Zang, and J. Xie, “Shadoweth: Private smart contract on public blockchain,” *Journal of Computer Science and Technology*, vol. 33, pp. 542–556, 2018.

- [33] M. Hajian Berenjestanaki, H. R. Barzegar, N. El Ioini, and C. Pahl, "Blockchain-based e-voting systems: a technology review," *Electronics*, vol. 13, no. 1, p. 17, 2023.
- [34] Y. Abuidris, R. Kumar, and W. Wenyong, "A survey of blockchain based on e-voting systems," in *Proceedings of the 2019 2nd International Conference on Blockchain Technology and Applications*, 2019, pp. 99–104.
- [35] T. Nguyen and M. T. Thai, "zvote: A blockchain-based privacy-preserving platform for remote e-voting," in *ICC 2022-IEEE International Conference on Communications*. IEEE, 2022, pp. 4745–4750.
- [36] H. Alamleh and A. A. S. AlQahtani, "Analysis of the design requirements for remote internet-based e-voting systems," in *2021 IEEE World AI IoT Congress (AIIoT)*. IEEE, 2021, pp. 0386–0390.
- [37] S. S. Chaeikar, A. Jolfaei, N. Mohammad, and P. Ostovari, "Security principles and challenges in electronic voting," in *2021 IEEE 25th International Enterprise Distributed Object Computing Workshop (EDOCW)*. IEEE, 2021, pp. 38–45.
- [38] A. Benabdallah, A. Audras, L. Coudert, N. El Madhoun, and M. Badra, "Analysis of blockchain solutions for e-voting: a systematic literature review," *IEEE Access*, vol. 10, pp. 70746–70759, 2022.
- [39] M. Hajian Berenjestanaki, H. R. Barzegar, N. El Ioini, and C. Pahl, "An investigation of scalability for blockchain-based e-voting applications," in *International Congress on Blockchain and Applications*. Springer, 2023, pp. 134–143.
- [40] Ethereum Foundation, "Eip-196: Precompiled contracts for addition and scalar multiplication on the elliptic curve alt_bn128," <https://eips.ethereum.org/EIPS/eip-196>, 2017, accessed: 2025-09-21.
- [41] Certicom Research, "Sec 2: Recommended elliptic curve domain parameters," <https://www.secg.org/sec2-v2.pdf>, 2010, accessed: 2025-09-21.
- [42] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Keccak," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2013, pp. 313–314.
- [43] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE transactions on information theory*, vol. 31, no. 4, pp. 469–472, 1985.
- [44] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," in *2018 IEEE symposium on security and privacy (SP)*. IEEE, 2018, pp. 315–334.
- [45] C.-P. Schnorr, "Efficient identification and signatures for smart cards," in *Conference on the Theory and Application of Cryptology*. Springer, 1989, pp. 239–252.
- [46] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)," *International journal of information security*, vol. 1, no. 1, pp. 36–63, 2001.
- [47] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct {Non-Interactive} zero knowledge for a von neumann architecture," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 781–796.
- [48] A. Banerjee, M. Clear, and H. Tewari, "zkhawk: Practical private smart contracts from mpc-based hawk," in *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*. IEEE, 2021, pp. 245–248.
- [49] J. Groth and M. Kohlweiss, "One-out-of-many proofs: Or how to leak a secret and spend a coin," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 253–280.
- [50] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu, "Zexe: Enabling decentralized private computation," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 947–964.
- [51] J. Groth and M. Maller, "Snarky signatures: Minimal signatures of knowledge from simulation-extractable snarks," in *Annual International Cryptology Conference*. Springer, 2017, pp. 581–612.
- [52] A. L. Xiong, B. Chen, Z. Zhang, B. Bünz, B. Fisch, F. Krell, and P. Camacho, "VeriZexe: Decentralized private computation with universal setup," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4445–4462.
- [53] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge," *Cryptology ePrint Archive*, 2019.
- [54] 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*, 2022, pp. 2735–2749.
- [55] R. Del Pino, V. Lyubashevsky, and G. Seiler, "Short discrete log proofs for fhe and ring-lwe ciphertexts," in *IACR International Workshop on Public Key Cryptography*. Springer, 2019, pp. 344–373.
- [56] G. Jeong, N. Lee, J. Kim, and H. Oh, "Azeroth: Auditable zero-knowledge transactions in smart contracts," *IEEE Access*, vol. 11, pp. 56463–56480, 2023.
- [57] J. Groth, "On the size of pairing-based non-interactive arguments," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2016, pp. 305–326.
- [58] Z. L. Jiang, M. Xie, H. Chen, Y. Pan, J. Lyu, M. H. Au, J. Fang, Y. Liu, and X. Wang, "Rpsec: Regulatable privacy-preserving smart contracts on account-based blockchain," *IEEE Transactions on Network Science and Engineering*, vol. 11, no. 5, pp. 4822–4835, 2024.
- [59] M. Bowman, A. Miele, M. Steiner, and B. Vavala, "Private data objects: an overview," *arXiv preprint arXiv:1807.05686*, 2018.
- [60] P. Das, L. Ekey, T. Frassetto, D. Gens, K. Hostáková, P. Jauernig, S. Faust, and A.-R. Sadeghi, "{FastKitten}: Practical smart contracts on bitcoin," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 801–818.
- [61] Y. Yan, C. Wei, X. Guo, X. Lu, X. Zheng, Q. Liu, C. Zhou, X. Song, B. Zhao, H. Zhang *et al.*, "Confidentiality support over financial grade consortium blockchain," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2227–2240.
- [62] C. Müller, M. Brandenburger, C. Cachin, P. Felber, C. Göttel, and V. Schiavoni, "Tz4fabric: Executing smart contracts with arm trustzone:(practical experience report)," in *2020 International symposium on reliable distributed systems (SRDS)*. IEEE, 2020, pp. 31–40.
- [63] Z. Jian, Y. Lu, Y. Qiao, Y. Fang, X. Xie, D. Yang, Z. Zhou, and T. Li, "Tsc-vee: A trustzone-based smart contract virtual execution environment," *IEEE Transactions on Parallel and Distributed Systems*, 2023.

Appendix A. Proof of Protocol Security

We prove that $\Pi_{\text{PrivContract}}(\lambda, S, \{\mathcal{P}_i\}_{i \in [n]})$ UC-realizes the ideal functionality $\mathcal{F}_{\text{PrivContract}}(\lambda, \ell, \{\mathcal{P}_i\}_{i \in [n]})$ with respect to a leakage function $\ell(x)$ that only reveals the length of x , i.e. $\ell(x) = 0^{|x|}$. In the protocol, $\ell(\cdot)$ is realized with IND-CPA encryption schemes.

Proof. Let $\mathcal{Z}$ be an environment and let $\mathcal{A}$ be a "dummy adversary" [12]. The dummy adversary relays messages between $\mathcal{Z}$ and clients. To show that $\Pi_{\text{PrivContract}}$ UC-realizes $\mathcal{F}_{\text{PrivContract}}$, we specify a simulator $\mathcal{S}$. The simulator $\mathcal{S}$ ensures that no environment can distinguish the interactions with $\Pi_{\text{PrivContract}}$ and $\mathcal{A}$ from the interactions with $\mathcal{F}_{\text{PrivContract}}$ and $\mathcal{S}$. That is, $\mathcal{S}$ must satisfy:

$$\forall \mathcal{Z}, \text{EXEC}_{\Pi_{\text{PrivContract}}, \mathcal{A}, \mathcal{Z}} \approx \text{EXEC}_{\mathcal{F}_{\text{PrivContract}}, \mathcal{S}, \mathcal{Z}}.$$

Ideal functionality $\mathcal{F}_{\text{PrivContract}}(\lambda, \ell, \{\mathcal{P}_i\}_{i \in [n]})$

- 1: **Parameter:** leakage function $\ell : \{0, 1\}^* \rightarrow \{0, 1\}^*$
- 2: **Init:** $\text{Storage} \leftarrow \emptyset, T \leftarrow \emptyset$
- 3: **On receive** (create, Contract) **from** $\mathcal{P}_i$ **for some** $i \in [n]$
- 4: $\text{cid} \xleftarrow{R} \mathcal{H}(\text{Contract})$
- 5: Notify $\mathcal{A}$ of (create, $\mathcal{P}_i$, cid, Contract); block until $\mathcal{A}$ replies
- 6: $\text{Storage}[\text{cid}] \leftarrow \text{Contract}$
- 7: Send a public delayed output (receipt, cid) to $\mathcal{P}_i$.
- 8: **On receive** (Init, cid, sid, prog) **from** $\mathcal{P}_i$ **for some** $i \in [n]$
- 9: Parse Contract $\leftarrow \text{Storage}[\text{cid}]$. Abort if not found.
- 10: Notify $\mathcal{A}$ of (Init, $\mathcal{P}_i$, cid, sid, $\ell(\text{prog})$); block until $\mathcal{A}$ replies.
- 11: Set $T[\mathcal{P}_i] := (\text{sid}, \text{prog}, 0)$.
- 12: Execute the contract and output $((\text{addr}, \text{report}), \text{mem})$.
- 13: Set $T[\mathcal{P}_i] := (\text{sid}, \text{prog}, \text{mem})$.
- 14: Notify $\mathcal{A}$ of (sid, $\ell(\text{addr}, \text{report}), \ell(\text{mem})$); block until $\mathcal{A}$ replies.
- 15: Send a public delayed output (ok, addr) to $\mathcal{P}_i$.
- 16: **On receive** (execute, cid, sid, eid, task_input) **from** $\mathcal{P}_i$ **for some** $i \in [n]$
- 17: Parse Contract $\leftarrow \text{Storage}[\text{cid}]$. Abort if not found.
- 18: Notify $\mathcal{A}$ of (execute, $\mathcal{P}_i$, cid, sid, eid, $\ell(\text{task_input})$); block until $\mathcal{A}$ replies.
- 19: Let $(\text{sid}', \text{prog}, \text{mem}) := T[\mathcal{P}_i]$. Abort if $\text{sid}' \neq \text{sid}$.
- 20: Execute the contract and output (outp, mem').
- 21: Notify $\mathcal{A}$ of (sid', $\ell(\text{outp}), \ell(\text{mem}')$); block until $\mathcal{A}$ replies.
- 22: Set $T[\text{eid}, \mathcal{P}_i] := (\text{sid}', \text{prog}, \text{mem}')$.
- 23: Generate a task ID taskid and send a secret delayed output (ok, taskid) to $\mathcal{P}_i$.

Figure 13: Ideal functionality of HyPSC protocol.

The general procedure for $\mathcal{S}$ is as follows. When a message is sent by an honest party to $\mathcal{F}_{\text{PrivContract}}$, $\mathcal{S}$ simulates the corresponding real-world “network traffic” for $\mathcal{Z}$, using information provided by $\mathcal{F}_{\text{PrivContract}}$. If a message originates from a corrupted party, $\mathcal{S}$ extracts the input from this message. The simulator $\mathcal{S}$ then interacts with the corrupted party, using the help of $\mathcal{F}_{\text{PrivContract}}$ to complete the simulation. We provide further details on the processing of specific messages.

Simulate the honest client $\mathcal{P}_i$:

• **Contract creation phase.**

- If the environment $\mathcal{Z}$ sends (create, Contract) to the dummy client $\hat{\mathcal{P}}_i$, $\hat{\mathcal{P}}_i$ forwards the message to the functionality $\mathcal{F}_{\text{PrivContract}}$.
- If $\mathcal{F}_{\text{PrivContract}}$ receives the message from $\hat{\mathcal{P}}_i$, $\mathcal{F}_{\text{PrivContract}}$ notifies the simulator $\mathcal{S}$ of the message (create, $\mathcal{P}_i$, cid, Contract).
- The simulator $\mathcal{S}$ internal simulates the real world $\Pi_{\text{PrivContract}}$ as follows:
 - a. First, the simulator $\mathcal{S}$ simulates $\mathcal{P}_i$ to invoke create(Contract) of $\mathcal{F}_{\text{Blockchain}}$.
 - b. Second, if the adversary forwards the message, $\mathcal{S}$ returns (receipt, cid) to $\mathcal{P}_i$ on behalf of $\mathcal{F}_{\text{Blockchain}}$.

• **Initialization phase.**

- If the environment $\mathcal{Z}$ sends (Init, cid, sid, prog) to the dummy client $\hat{\mathcal{P}}_i$, $\hat{\mathcal{P}}_i$ forwards the message to the functionality $\mathcal{F}_{\text{PrivContract}}$.
- If $\mathcal{F}_{\text{PrivContract}}$ receives the message from the dummy client $\hat{\mathcal{P}}_i$, $\mathcal{F}_{\text{PrivContract}}$ notifies the simulator $\mathcal{S}$ of the message (Init, $\mathcal{P}_i$, cid, sid, $\ell(\text{prog})$).
- Then, the functionality $\mathcal{F}_{\text{PrivContract}}$ updates the state, notifies the simulator $\mathcal{S}$ the message (sid', $\ell(\text{addr}), \ell(\text{mem}')$) as shown in Figure 13 and blocks.
- The simulator $\mathcal{S}$ internal simulates the real world $\Pi_{\text{PrivContract}}$ as follows:
 - a. First, the simulator $\mathcal{S}$ generates a random string of length $\ell(\text{prog})$, denoted prög.
 - b. Next, the simulator $\mathcal{S}$ simulates $\mathcal{P}_i$ to invoke install(sid, prög) of $\mathcal{G}_{\text{att}}$.
 - c. Then, the simulator $\mathcal{S}$ generates $\text{eid} \xleftarrow{R} \{0, 1\}^\lambda$ and returns (sid, eid) on behalf of $\mathcal{G}_{\text{att}}$.
 - d. Next, the simulator $\mathcal{S}$ simulates $\mathcal{P}_i$ to invoke getpk(sid) of $\mathcal{G}_{\text{att}}$.
 - e. Then, the simulator $\mathcal{S}$ generates the key pair $\text{pk}_{\text{TEE}}, \text{sk}_{\text{TEE}} \leftarrow \{0, 1\}^*$ and returns pk_{TEE} to $\mathcal{P}_i$ on behalf of $\mathcal{G}_{\text{att}}$.
 - f. Next, the simulator $\mathcal{S}$ simulates $\mathcal{P}_i$ to invoke resume(sid, eid, (Init, pk_{TEE})) of $\mathcal{G}_{\text{att}}$.
 - g. Then, the simulator $\mathcal{S}$ generates a random address $\hat{\text{addr}} \leftarrow \{0, 1\}^{\ell(\text{addr})}$ and computes the signature $\hat{\sigma} := \text{S.Sign}_{\text{sk}_{\text{TEE}}}(\text{sid}, \text{eid}, \text{prög}, \hat{\text{addr}})$. The simulator $\mathcal{S}$ returns the message (sid, $\hat{\sigma}$, $\hat{\text{addr}}$) to $\mathcal{P}_i$ on behalf of $\mathcal{G}_{\text{att}}$.
 - h. Next, the simulator $\mathcal{S}$ simulates client $\mathcal{P}_i$ to invoke register(cid, addr, pk_{TEE} , $\hat{\sigma}$) of $\mathcal{F}_{\text{Blockchain}}$.
 - i. Finally, if the adversary forwards the message, the simulator $\mathcal{S}$ returns (ok, addr) to the $\mathcal{P}_i$ on behalf of $\mathcal{F}_{\text{Blockchain}}$.

• **Execute phase.**

- If the environment $\mathcal{Z}$ sends (execute, cid, sid, eid, task_input) to the dummy client $\hat{\mathcal{P}}_i$, $\hat{\mathcal{P}}_i$ forwards the message to the functionality $\mathcal{F}_{\text{PrivContract}}$.
- If $\mathcal{F}_{\text{PrivContract}}$ receives the message from the dummy client $\hat{\mathcal{P}}_i$, $\mathcal{F}_{\text{PrivContract}}$ notifies the simulator $\mathcal{S}$ of the message (execute, $\mathcal{P}_i$, cid, sid, eid, $\ell(\text{task_input})$).
- Then, the functionality $\mathcal{F}_{\text{PrivContract}}$ updates the state, notifies the simulator $\mathcal{S}$ the message (sid', $\ell(\text{outp}), \ell(\text{mem}')$) as shown in Figure 13 and blocks.
- The simulator $\mathcal{S}$ internal simulates the real world $\Pi_{\text{PrivContract}}$ as follows:
 - a. First, $\mathcal{S}$ generates a random input $\text{task_input} \leftarrow \{0, 1\}^{\ell(\text{task_input})}$ and invokes resume(sid, eid, (execute, task_input)) of $\mathcal{G}_{\text{att}}$.
 - b. Then, $\mathcal{S}$ generates an IND-CPA ciphertext outp of a random string with the same length as the real message and computes the signature $\hat{\sigma} := \text{S.Sign}_{\text{sk}_{\text{TEE}}}(\text{sid}, \text{eid}, \text{prög}, \text{outp})$. The simulator $\mathcal{S}$ and returns the message (sid, $\hat{\sigma}$, outp) to $\mathcal{P}_i$ on

behalf of $\mathcal{G}_{\text{att}}$.

- c. Next, $\mathcal{S}$ simulates $\mathcal{P}_i$ to invoke $\text{execute}(cid, \hat{outp}, \hat{\sigma})$ of $\mathcal{F}_{\text{Blockchain}}$.
- d. Finally, if the adversary forwards the message, $\mathcal{S}$ returns $(\text{ok}, \text{taskid})$ to $\mathcal{P}_i$ on behalf of $\mathcal{F}_{\text{Blockchain}}$.

Simulate the corrupted client $\mathcal{P}_i$:

• Contract creation phase.

- When the environment $\mathcal{Z}$ indicates $\mathcal{P}_i$ to create a contract and $\mathcal{P}_i$ invokes $(\text{create}, \text{Contract})$ of $\mathcal{F}_{\text{Blockchain}}$.
- The simulator extracts the contract Contract and sends $(\text{create}, \text{Contract})$ to $\mathcal{F}_{\text{PrivContract}}$ on behalf of the dummy client $\hat{\mathcal{P}}_i$.
- If $\mathcal{F}_{\text{PrivContract}}$ receives the message from $\hat{\mathcal{P}}_i$, $\mathcal{F}_{\text{PrivContract}}$ notifies the simulator $\mathcal{S}$ of the message $(\text{create}, \hat{\mathcal{P}}_i, cid, \text{Contract})$ and blocks.
- Finally, if the adversary forwards the message, $\mathcal{S}$ returns $(\text{receipt}, cid)$ to $\mathcal{P}_i$ on behalf of $\mathcal{F}_{\text{Blockchain}}$.

• Initialization phase.

- When the environment $\mathcal{Z}$ indicates $\mathcal{P}_i$ to initialize a program and $\mathcal{P}_i$ invokes $(\text{Init}, cid, sid, \text{prog})$ of the blockchain nodes.
- When $\mathcal{P}_i$ invokes the $\text{install}(sid, \text{prog})$ of the $\mathcal{G}_{\text{att}}$, the simulator extracts the program prog and sends $(\text{Init}, cid, sid, \text{prog})$ to the $\mathcal{F}_{\text{PrivContract}}$ on behalf of the dummy client $\hat{\mathcal{P}}_i$.
- If $\mathcal{F}_{\text{PrivContract}}$ receives the message from the dummy client $\hat{\mathcal{P}}_i$, $\mathcal{F}_{\text{PrivContract}}$ notifies the simulator $\mathcal{S}$ of the message $(\text{Init}, \hat{\mathcal{P}}_i, cid, sid, \ell(\text{prog}))$.
- Then, the functionality $\mathcal{F}_{\text{PrivContract}}$ updates the state, notifies the simulator $\mathcal{S}$ the message $(sid', \ell(addr), \ell(mem'))$ as shown in Figure 13 and blocks.
- The simulator $\mathcal{S}$ executes install on behalf of $\mathcal{G}_{\text{att}}$ in Figure 2 and returns (sid, eid) .
- If $\mathcal{P}_i$ invokes $\text{getpk}(sid)$ of $\mathcal{G}_{\text{att}}$, the simulator returns pk_{TEE} to $\mathcal{P}_i$ on behalf of $\mathcal{G}_{\text{att}}$.
- If $\mathcal{P}_i$ invokes $\text{resume}(sid, eid, (\text{Init}, \text{pk}_{TEE}))$ of $\mathcal{G}_{\text{att}}$, the simulator generates a random address $addr \leftarrow \{0, 1\}^{\ell(addr)}$ and computes the signature $\sigma := \text{S.Sign}_{\text{sk}_{TEE}}(sid, eid, \text{prog}, addr)$. The simulator $\mathcal{S}$ returns the message $(sid, \sigma, addr)$ to $\mathcal{P}_i$ on behalf of $\mathcal{G}_{\text{att}}$.
- If $\mathcal{P}_i$ invokes the $\text{register}(cid, \hat{addr}, \text{pk}_{TEE}, \hat{\sigma})$ to $\mathcal{F}_{\text{Blockchain}}$. The simulator checks whether $\hat{\sigma} = \sigma$ and $addr = \hat{addr}$. If check false, the simulator aborts.
- Finally, if the adversary forwards the message, the simulator returns $(\text{ok}, addr)$ to $\mathcal{P}_i$ on behalf of $\mathcal{F}_{\text{Blockchain}}$.

• Execution phase:

- When the environment $\mathcal{Z}$ indicates $\mathcal{P}_i$ to execute the program and $\mathcal{P}_i$ invokes $\text{resume}(sid, eid, (\text{execute}, \text{task_input}))$ of $\mathcal{G}_{\text{att}}$
- Then, the simulator extracts the input task_input and sends $(\text{execute}, cid, sid, eid, \text{task_input})$ to $\mathcal{F}_{\text{PrivContract}}$ on behalf of the dummy client $\hat{\mathcal{P}}_i$.
- If $\mathcal{F}_{\text{PrivContract}}$ receives the message from the dummy client $\hat{\mathcal{P}}_i$, $\mathcal{F}_{\text{PrivContract}}$ notifies the simulator $\mathcal{S}$ of the message $(\text{execute}, \hat{\mathcal{P}}_i, cid, sid, eid, \ell(\text{task_input}))$.

- Then, the functionality $\mathcal{F}_{\text{PrivContract}}$ updates the state, notifies the simulator $\mathcal{S}$ the message $(sid', \ell(outp), \ell(mem'))$ as shown in Figure 13 and blocks.
- The simulator generates an IND-CPA ciphertext $\hat{outp}$ of a random string with the same length as the real message and computes the signature $\hat{\sigma} := \text{S.Sign}_{\text{sk}_{TEE}}(sid, eid, \text{prog}, \hat{outp})$. The simulator $\mathcal{S}$ returns the message $(sid, \hat{\sigma}, \hat{outp})$ to $\mathcal{P}_i$ on behalf of $\mathcal{G}_{\text{att}}$.
- If $\mathcal{P}_i$ invokes the $\text{execute}(cid, outp, \hat{\sigma})$ to $\mathcal{F}_{\text{Blockchain}}$. The simulator checks whether $\hat{\sigma} = \sigma$. If check false, the simulator aborts.
- Finally, if the adversary forwards the message, the simulator returns $(\text{ok}, \text{taskid})$ to $\mathcal{P}_i$ on behalf of $\mathcal{F}_{\text{Blockchain}}$.

We show that no environment $\mathcal{Z}$ can distinguish the real protocol $\Pi_{\text{PrivContract}}$ from the ideal world in the $(\mathcal{G}_{\text{att}}, \mathcal{F}_{\text{Blockchain}})$ -hybrid model by a sequence of hybrid arguments:

Hybrid H_0 : This is the real protocol $\text{EXEC}_{\Pi_{\text{PrivContract}}, \mathcal{A}, \mathcal{Z}}$.

Hybrid H_1 : H_1 proceeds as in the real world protocol, except that $\mathcal{S}$ simulates $\mathcal{G}_{\text{att}}$ and $\mathcal{F}_{\text{Blockchain}}$. Specifically, $\mathcal{S}$ generates a key pair $(\text{pk}_{TEE}, \text{sk}_{TEE})$ for $\mathcal{G}_{\text{att}}$ and executes on behalf of $\mathcal{G}_{\text{att}}$. Whenever $\mathcal{A}$ wants to communicate with $\mathcal{G}_{\text{att}}$, $\mathcal{S}$ records the messages of $\mathcal{A}$ and honestly simulates the behavior of $\mathcal{G}_{\text{att}}$. Similarly, $\mathcal{S}$ simulates $\mathcal{F}_{\text{Blockchain}}$ by storing the states of clients internally.

As the view of $\mathcal{A}$ in H_1 is perfectly simulated as in the real world, $\mathcal{Z}$ cannot distinguish between H_1 and the real execution H_0 .

Hybrid H_2 : H_2 proceeds as in H_1 , except for the following modifications. If $\mathcal{A}$ invoked $\mathcal{F}_{\text{Blockchain}}$ with a message $\text{register}(cid, \hat{addr}, \text{pk}_{TEE}, \hat{\sigma})$ or $\text{execute}(cid, \hat{task}, \hat{\sigma})$, the simulator $\mathcal{S}$ checks whether $\hat{\sigma}$ is the signature it generated on behalf of $\mathcal{G}_{\text{att}}$. If this check fails, $\mathcal{S}$ outputs abort.

The indistinguishability between H_1 and H_2 can be shown by the following reduction to the EU-CMA property of the signature scheme S . Sepically in H_1 , if $\mathcal{A}$ sends forged signatures to $\mathcal{F}_{\text{Blockchain}}$, signature verification by $\mathcal{F}_{\text{Blockchain}}$ will fail with all but negligible probability. If $\mathcal{Z}$ can distinguish H_2 from H_1 , $\mathcal{Z}$ and $\mathcal{A}$ can be used to win the game of signature forgery.

Hybrid H_3 : H_3 proceeds as in H_2 , $(sid, \sigma, outp)$ sent from $\mathcal{G}_{\text{att}}$ to $\mathcal{P}_i$ with task in H_2 is replaced with $(sid, \hat{\sigma}, \hat{task})$, where $\hat{task}$ is an IND-CPA ciphertext of a random string with the same length as the real message. Indistinguishability between H_5 and H_4 can be reduced to the IND-CPA property of PKE. Without the secret key, $\mathcal{A}$ cannot distinguish the encryption of a pseudorandom string from the encryption of other messages.

Across **Hybrid $H_0 - H_3$** , we have $H_0 = H_1 \approx H_2 \approx H_3$, which finishes the proof. $\square$