Phala Network: A Confidential Smart Contract Network Based on Polkadot

Hang Yin Shunfan Zhou Jun Jiang

October 17, 2019

1 Introduction

Nowadays the security of the permissionless blockchain is typically guaranteed by state replication over consensus algorithms. Though this approach works well for blockchain, it also means that everything on the blockchain is *public*, which brings a problem: confidential information cannot be handled by the blockchain. The lack of confidentiality greatly limits the usage of blockchain in processing sensitive business and user data. For example, stock traders usually do not want to reveal their positions or order history. What's more, all of the privacy-related DApps on the Ethereum cannot comply with the General Data Protection Regulation and thus will be prevented in European Union.

Several methodologies have been proposed to address the privacy problem. Monero and Zcash implemented private transaction by ring signature [13] and zk-SNARK [11] technology, but their methods can only provide privacy for cryptocurrencies and are hard to be extended to general-purpose smart contracts. MPC (Multi-Party Computing) can theoretically run arbitrary programs without revealing intermediate states to the participants, with the expense of a performance overhead of 10^6 times [9], which makes it impractical for real world use cases.

A new approach is to utilize special hardware, i.e., Trusted Execution Environment (TEE) [8] . TEE is a special area in some processors that provides a higher level of security including isolated execution, code integration, and state confidentiality. Naive TEE as a computing platform has several shortages, such as the lack of a reliable time source and availability guarantee.

Ekiden [9] fixed these problems by introducing a TEE-blockchain hybrid architecture and implemented high performance confidential smart contract platform. However, contracts in Ekiden are isolated, which means the contracts cannot interoperate with each other, let alone external blockchains. Interoperability is a keystone of modern smart contracts. For example, 8 of the top 10 contracts in Ethereum, the largest smart contract platform in the world, rely on functions invocations or money transferring to at least one other contract. Without interoperability, contracts cannot read information or call functions from other smart contracts. What's more, the use of self-defined tokens, one

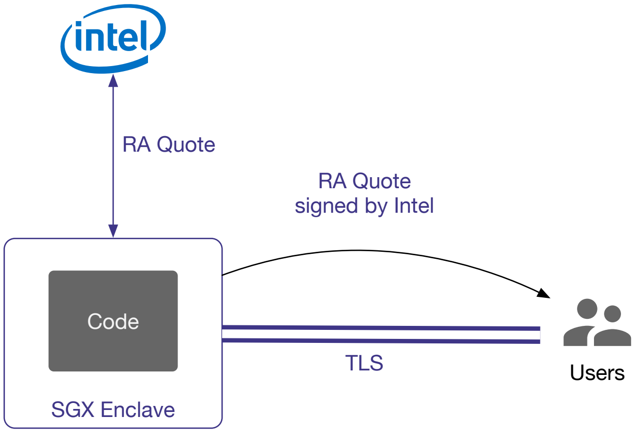

Figure 1: Intel SGX remote attestation procedure.

of the most common cases in smart contract usages, is unachievable if all the contracts cannot access the token contract.

In this paper, we present Phala Network, a novel cross-chain interoperable confidential smart contract network as a Polkadot parachain [10]. We introduce an *Event Sourcing / Command Query Responsibility Segregation* [4, 3] architecture into a TEE-blockchain hybrid system to achieve cross-contract and cross-chain interoperability for confidential smart contracts. We further designed a Libra-Polkadot bridge to implement a privacy-preserving Libra Coin by confidential contract.

2 Background

2.1 Intel SGX: A TEE Implementation

Intel SGX [6] is a popular implementation of TEE. It runs code inside a special "Enclave" so that the execution of the code is deterministic, i.e., not affected by other processes or underlying operating system, and the intermediate states is not leaked. In a properly set up system, Intel SGX can defend the attacks from the OS layer and hardware layer.

To ensure the execution is finished as expected inside an enclave, a proof can be generated according to a protocol called **Remote Attestation**. The hardware can generate an *attestation quote* based on the details of hardware, firmware, the code being executed inside the enclave, and other user-defined data produced by the code. The quote is signed by the trusted hardware with credentials embedded during the production process.

Next, the generated attestation quote is sent to the Intel Remote Attestation Service. Intel will sign the quote iff the signing credentials are valid. As each credential is uniquely bound to an Intel CPU unit, fake attestation quotes will never pass the Remote Attestation Service check.

Finally, the attestation quote signed by Intel serves as the proof of a successful execution. It proves that specific code has been run inside an SGX enclave and produces certain output, which implies the confidentiality and the correctness of the execution. The proof can be published and validated by anyone with generic hardware.

Intel SGX and the Remote Attestation protocol is the foundation of confidential contract. Except for Intel SGX, there are also alternative implementation choices like AMD SEV [1] and ARM TrustZone [2].

2.2 Event Sourcing and CQRS

Event Sourcing is a software design pattern. Instead of storing the latest state of the data, the events causing state transition are recorded in an append-only log. The events are timestamped and can be replayed to reconstruct the state of any time. Since the timestamped events, the state of the system is deterministic. Command Query Responsibility Segregation (CQRS) is a design pattern by which the read operations and write operations are handled separately. In a CQRS and Event Sourcing combined system, the write operations are recorded as the events and the read operations can be served by the current view of the state. This pattern make a system easy to scale up and avoid conflicts.

For native CPU performance and better security, a single or a small set of TEEs are bound to each confidential contract to execute it. By this design, the contract state is isolated from each other without consistency guarantee. It becomes a trouble for cross-contract and even cross-chain interoperability.

While keeping a strong state consistency is hard, contracts can still communicate by passing messages on the premise that state transition is still deterministic. In an Event Sourcing / CQRS design, the commands can be initiated from users or contracts and are strictly timestamped on the blockchain. It guarantees the global state is deterministic and therefore enables message passing between contracts. Message passing is a primitive to implement higher level interoperability like contract invocation and token transferring. The read-only operations are not timestamped for better performance.

3 Confidential Contract

Phala Network aims to build a platform for general-purpose privacy-preserving Turing-Complete smart contracts. The basic requirements for such a platform could be as follows.

- **Confidentiality.** Unlike the existing blockchains for smart contracts, Phala Network avoids the leakage of any input, output, or intermediate state of confidential contract. Only authorized queries to the contract will be answered.
- **Code Integrity.** Anyone can verify that an output is produced by a specific smart contract published on the blockchain.

- **State Consistency.** Anyone can verify that an execution happened at a certain blockchain height, which implies the output of the execution is subject to a certain chain state.
- **Availability.** There must not be a single point of failure such as disconnection of the miner.
- **Interoperability.** Contracts can interoperate with each other and external blockchains.

The existing TEE solutions, e.g., Intel SGX, can only prevent the leakage of sensitive information during the execution of *isolated* programs, and provide no guarantee on availability or verification of input data. Thus it requires a carefully-designed infrastructure to integrate TEE into blockchain to meet the requirements above.

We are going to introduce the design of Phala Network and how it fulfills the above requirements in the following sections.

3.1 Abstraction of Confidential Contract

A typical smart contract can be regarded as a state machine of a current state s_n and a state transition function f, which takes input event e_n and last state s_{n-1} to produce the latest state s_n:

$$s_n = f(s_{n-1}, e_n)$$

Since the state transition process happens inside the enclave, any of its intermediate states remains invisible to outside. We can further encrypt the reached state and input event to prevent the attackers from inferring the internal state of contract with event replay.

Let cs_n be the cipher of s_n and ce_n be the cipher of e_n, the state transition function of a confidential contract p can be represented as:

$$cs_n = p(cs_{n-1}, ce_n) \tag{1}$$

$$p(cs_{n-1}, ce_n) = \mathrm{Enc}\Bigg(f\big(\mathrm{Dec}(cs_{n-1}), \mathrm{Dec}(ce_n)\big)\Bigg) \tag{2}$$

where Enc and Dec can be carefully-chosen symmetric encryption and decryption functions subject to the contract.

Unlike the existing smart contract, a confidential contract doesn't expose any information outside the enclave by default. To answer authorized queries, we introduce a query function q which takes the current encrypted state cs_n, query parameters $paras$ and user's identity I (usually a pubkey) as input and returns the response r:

$$r = q(cs_n, paras, I)$$

The confidential contract must first validate the identity of the user and then respond to her query. Apart from the queries from users, the contract may

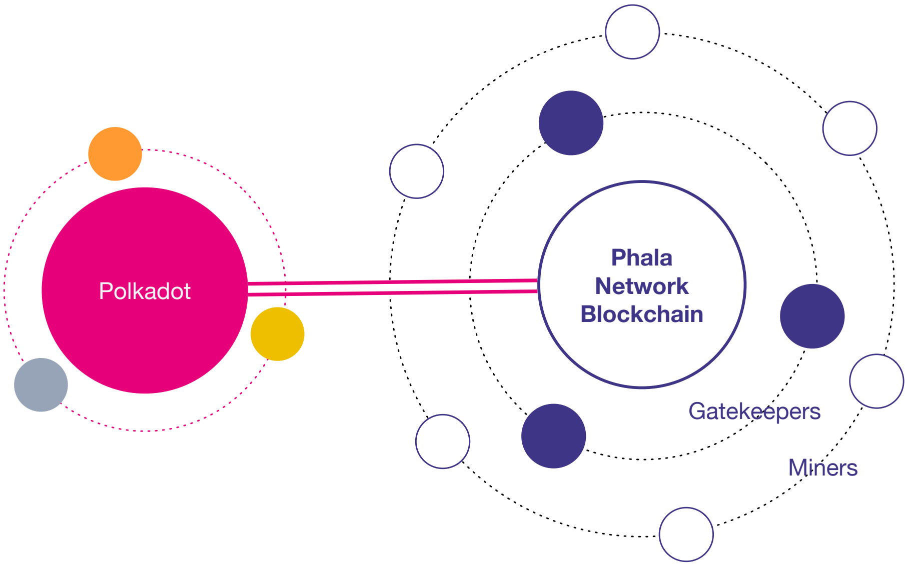

Figure 2: Roles in the protocol.

also accept a special query producing side effects. The side effects include the egressing data that can be posted back to the blockchain by miners.

4 The Protocol

There are a few roles involved in the protocol as shown in Figure 2.

- **Users.** Users invoke, query and deploy smart contracts. Users interact with smart contracts via **Blockchain** and **Worker Nodes**. They can verify the blockchain as well as the cryptographic evidence on the blockchain independently by running a light client or full node. No special hardware, i.e., TEE, is needed for users to use the confidential contracts.
- **Worker Nodes.** Worker Nodes run confidential contracts in TEE-compatible hardwares. Worker Nodes are off-chain. In each node, a special program called `pRuntime` is deployed to the enclave. The runtime has a builtin VM to run contracts. It also cooperates with the blockchain to support the contracts in full life cycle. Worker Nodes can be further divided into three roles:
  - **Genesis Node.** Genesis Node helps bootstrap the network and set up the cryptographic configuration. There's only one Genesis Node and it's destroyed after the launch of Phala Network.
  - **Gatekeepers.** Gatekeepers manage the secrets to ensure the availability and security of the network. Gatekeepers are dynamically elected on the blockchain and they stake a large amount of Phala token. They are rewarded for being online and slashed in case of misbehavior because there

must be a certain number of functional Gatekeepers running at any time.

  - **Miners.** Miners execute the confidential contracts. They get paid by providing their computing resources to the users. Unlike Gatekeepers, Miners only need to stake a small amount of the Phala token and can join and exit the network as they want.

- **Remote Attestation Service.** Remote Attestation Service is a public service to validate if a Worker Node has deployed `pRuntime` correctly. The cryptographic evidence produced by the service can prove a certain output is produced by `pRuntime` running inside a TEE. IAS [5] is Intel SGX's remote attestation service implementation.
- **Blockchain.** Blockchain is the backbone of Phala Network. It stores the identities of the Worker Nodes, the published confidential contracts, the encrypted contract state, and the invocation transactions from users and other blockchains. When plugged into a Polkadot parachain slot, it's capable to interoperate with other blockchains through the Polkadot relay chain.

4.1 System Design

We first propose an important property which works as the basis of the following system design.

All worker nodes are non-Byzantine nodes. As illustrated above, each worker node runs a special program `pRuntime` in its TEE. The `pRuntime`, as the name suggests, works as the runtime environment for confidential contracts. It exports a set of APIs for contracts to access the state resources, manages the connections to the blockchain and answers user queries securely. `pRuntime` implements the Phala Network protocol described in this paper. Since the integrity of `pRuntime` is guaranteed by the remote attestation service, this isolation promises that no Byzantine fault can happen unless both the TEE and `pRuntime` are compromised, and an adversarial worker node can only launch a Denial-of-Service attack, which can be further detected by our responsiveness monitoring protocol.

In our system, the executors, i.e., miners, are stateless, which means the latest state of a confidential contract has to be got by sequentially executing all the input events on the blockchain or from a cached contract state and the events after that. All the inputs have to be first published to the blockchain and in this way, the blockchain works as the canonical source of contract inputs, which implies *Event Sourcing* design pattern. We further utilized the idea of *CQRS* in the design of the protocol to accelerate the users' queries to the contracts.

`pRuntime` maintains a set of secrets durning its life cycle. Contract states are encrypted and checkpointed on the blockchain with a symmetric key. Each `pRuntime` registers its identity on the blockchain and establishes secure connections to users with an asymmetric key pair. Since each `pRuntime` registers itself on the blockchain, any user can validate its identity. These secrets never go outside the `pRuntime`.

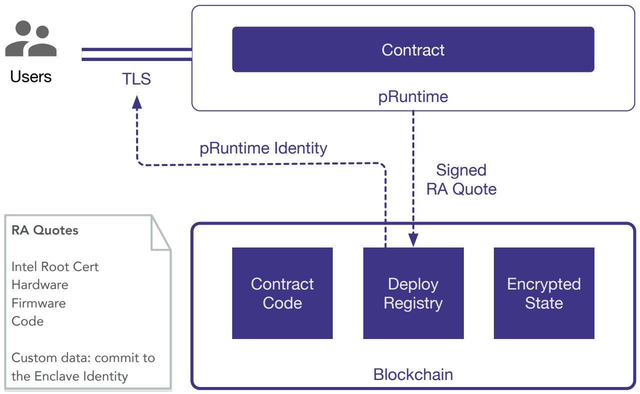

Figure 3: RA and communication.

4.2 Node Registration

All the Worker Nodes are required to be registered on the blockchain before participating in mining or Gatekeeper election.

Remote attestation provides a building block to verify the execution as well as its output of a certain code inside the enclave. However, running such attestation on each execution is time inefficient. In Phala Network we adopt a better protocol. The attestation measures the `pRuntime` instance and the generated unique identity during the registration, instead of each execution. In this way, a single attestation is sufficient to ensure the future behavior of the `pRuntime`.

1. The host program of the worker node n (i.e. Miner or Gatekeeper) calls `pRuntime.GetIdentity` to generate a key pair as an identity $I_n(pk_n, sk_n)$. sk_n is kept inside the enclave and pk_n is revealed to the host.
2. The host calls `pRuntime.GetRAQuote` to generate a remote attestation quote q with the commitment to pk_n and other necessary metadata.
3. The host submits q to Remote Attestation Service and get the signed quotes q_{signed}.
4. The host submits (q_{signed}, pk_n) to the blockchain. The blockchain then accepts and stores the information after validation.

The registered `pRuntime` instances are non-Byzantine. The `pRuntime` instance and the generated identity is measured by the attestation. It implies `pRuntime` runs in a TEE and the identity is generated securely. Since the privkey is unknown to any party expect the instance, nobody can pretend to be a registered `pRuntime`.

With the identity registered, a TLS-like channel between the requester and the target `pRuntime` can be established. The identities published on the blockchain serve as the PKI to avoid MitM attack. As the `pRuntime` is non-Byzantine in such a communication channel, the `pRuntime` can be trusted without further need of remote attestation. This trick improves the efficiency and flexibility of code execution in the runtime.

The metadata contains other necessary information, for example, the network endpoint of the node like libp2p multiaddrs. The registration has an expiration date to prevent vulnerabilities discovered in the future. Worker nodes can renew it by redo the registration process including generating a new identity and do the remote attestation.

4.3 State Encryption

Confidential contract runs on `pRuntime` and the states are persisted on the blockchain to ensure the availability. As the information on the blockchain are public, the saved states have to be encrypted. In Phala Network, each contract (c_i) is associated with a symmetric key called **Contract Key** (k_{c_i}).

Contract keys are generated by Gatekeepers inside `pRuntime`. To run a contract, a registered miner should get the corresponding contract key from Gatekeepers. The miner reads the latest contract states and decrypt it with the contract key during initialization. Then the updated states are encrypted and saved to the blockchain in the future. Gatekeepers hold the contract keys as a part of their states. Gatekeepers share a symmetric key called **Root Key** (k_r). The Gatekeeper states are also encrypted and saved to the blockchain like miners. All the keys are used by and kept in `pRuntime`.

The details of key management are discussed in below subsections. We further propose a few improvements including key rotation (in “Key Rotation” subsection) and a distributed key generation (DKG) scheme in “Open Questions” section.

4.4 Blockchain Launch

In the blockchain launch phase, the blockchain will load the initial distribution of the native token and start the election of Gatekeepers. A Genesis Node assists the launch of the blockchain.

1. Before the genesis block, the Genesis Node runs `pRuntime.Bootstrap` to generate a key pair as the identity of the node, and a symmetric key, namely Genesis Identity $I_g(pk_g, sk_g)$ and the genesis Root Key $k_r^{(0)}$. The runtime reveals pk_g but keeps sk_g and $k_r^{(0)}$ privately.

2. Start the blockchain with pk_g. In this stage pk_g is published in the genesis block and is used for other worker nodes to establish secure channels to the Genesis Node. $k_r^{(0)}$ is kept inside the Genesis Node and is used to store secrets necessary to run the network on the blockchain. The initial native token distribution is loaded in the genesis block.

3. The blockchain is at pre-launch phase after the genesis block. Governance module is enabled but other modules including confidential contract are still disabled until the Gatekeepers are elected.

4. Worker Nodes who want to participate in Gatekeeper election can follow the **Node Registration** scheme to register their identities on the blockchain. Then $n_{gatekeepers}$ (a chain parameter between tens to hundreds) Gatekeepers will be elected during the pre-launch phase. This can be done on-chain via an Polkadot-style NPoS validator election (see Appendix II for details).

5. When the election is finished, the Gatekeepers send a request to the Genesis Node for $k_r^{(0)}$ through a TLS connection. The Genesis Node only answers the requests from the selected Gatekeeper. Gatekeepers confirms their readiness the on the blockchain.

6. The Genesis Node retires and self destroys when it sees the confirmations of the Gatekeepers from the blockchain. In case there are unresponsive Gatekeepers, the election will expire and restart after the deadline.

So far Gatekeepers have their identity registered on the blockchain and $k_r^{(0)}$ has been distributed to all the Gatekeepers' runtime. Then $k_r^{(0)}$ will be used to deploy nodes and contracts in the future.

Periodical key rotation is necessary for forward secrecy. The root key at era n is denoted by $k_r^{(n)}$. We will discuss the details in "Key Rotation" subsection.

4.5 Deploy Worker Nodes

Both miners and Gatekeepers have to follow the "Node Registration" scheme to join the network. The protocol ensures the nodes are non-Byzantine. So we only consider responsiveness failures. To ensure the service quality, all the worker nodes must stake a certain amount of the Phala token and **could be slashed** once it fails to meet the responsive requirements. We will discuss the details about staking and monitoring in "Responsiveness Monitoring" section.

As Gatekeepers store the root key and need to be always online, they have to meet a higher standard and need to stake a larger amount. They are rewarded for keeping online and could be slashed otherwise.

4.6 Deploy the Contract

The bytecode of the compiled contract is published on the blockchain and then loaded by a user-specified miner to its `pRuntime`. The Gatekeepers generate a symmetric encryption key for each newly published contract. The key is shared with the corresponding `pRuntime` for state encryption. More specifically:

1. The developer publish the contract c_i to the blockchain

2. Once Gatekeepers notice the publish of c_i, they generate a corresponding contract key k_{c_i} for contract states encryption.

3. Gatekeepers save k_{c_i} to the blockchain as a part of the chainstate encrypted with Root Key $k_r^{(n)}$

4. The developer finds an available miner to load the contract. The developer can either run his miner so that no extra fee is needed, or rent one from a resource market (see "Economic Design Paper" for more details).

5. The miner runtime connects to Gatekeeper through a secure connection and asks for k_{c_i} by `pRuntime.GetContractKey`.

The miner's `pRuntime` will use k_{c_i} to encrypt the contract state and save it to the blockchain periodically. We will discuss the details in "Execute the Contract" and "State Recover" section.

4.7 Key Rotation

The Gatekeepers are re-elected in each era according to the election rule (Appendix II). While rotating the Gatekeepers, the root key is also rotated. Let G_{n-1} and G_n be the Gatekeepers set before and after the election at era n. The root key will be rotated as follows:

1. G_n is elected on-chain by the governance module in the last block of each era. Simultaneously two leaders $l_{n-1} \in G_{n-1}$ and $l_n \in G_n$ are randomly chosen. The randomness comes from the random beacon in the block.

2. Any communication between miners and Gatekeepers is blocked until the key rotation is done.

3. Once the block is finalized, l_{n-1} and l_n collaboratively generate the next root key $k_r^{(n)}$ by a DH-like key exchange protocol with salts. l_{n-1} decrypts the Gatekeeper states and re-encrypt with $k_r^{(n)}$ and save it to the blockchain. l_n broadcasts $k_r^{(n)}$ to other members in G_n by a secure communication channel. G_n members will confirm the readiness on-chain when they received $k_r^{(n)}$.

4. Once all the G_n members have confirmed and l_{n-1} has updated the Gatekeeper states, the blockchain will generate a "Key Rotation Ready" event so that miners can resume the communication with the new Gatekeepers. Simultaneously the G_{n-1} members who are not still in G_n will notice the event and perform self destruction.

5. There's an expiration time for the key rotation procedure in case of the unresponsiveness. In such case, the unresponsive Gatekeepers will be slashed and the procedure should start over.

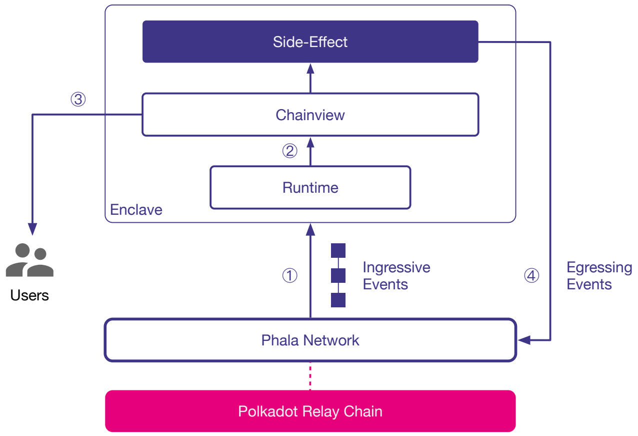

Figure 4: Execute the contract.

All the off-chain key management and encryption is handled by the `pRuntime` of the Gatekeepers.

The key rotation protocol starts from the first block in an era and ends after the final on-chain confirmation. Durning this period no new contract can be deployed to the miners because the communication between miners and Gatekeepers is blocked. However the delay caused by the protocol is trivial compared with the length of an era. In the best case, the delay is 2 round-trip-time (two on-chain confirmations).

Contract keys can be rotated in a simpler way. The miner generates a new contract key and re-encrypt the states with the new key. Then it can send the new contract key to Gatekeepers in a secure channel, and save the states to the blockchain. The two actions are combined in a single transaction to make then atomic.

Root key and contract key rotation ensures the forward secrecy of the confidential state (for both Gatekeeper state and contracts state). The keys for the obsolete data are destroyed once the rotation is done.

4.8 Execute the Contract

We adopt an Event Sourcing / CQRS style architecture for the contract execution. Read queries and write commands are segregated.

The contract state is determined by the write commands which have multiple sources: user invocations, blockchain events, and ingressive messages from the relay chain. In a naive design, we ask all the write commands to be recorded explicitly on the blockchain. The commands are denoted by **Ingressive Events** to `pRuntime`. As the events on the blockchain are ordered naturally, the blockchain becomes a canonical source of events.

1. The miner (host) ingests the ingressive events to the runtime by calling

`pRuntime.SyncBlockchain`. The incoming events are paired with cryptographic evidence which is validated by a light-weight Substrate client inside `pRuntime`. It ensures the integrity of the incoming events.

2. The runtime picks out the events targeting the contract deployed inside and feeds the contract. The execution of the contract produces a view of the `Chainstate`, namely `Chainview`

3. At any time, users can query `Chainview` by `pRuntime.Query`. The identity of the caller is attached to the queries so that the contract can decide whether to answer or not based on its customized authorization policy. The response to the query includes the query result as well as a commitment to the current blockchain state (e.g. height) and the contract state. In other words, all the outputs are subject to a certain blockchain state.

4. The runtime also produces various side-effects which is accessible by `pRuntime.DumpSideEffect`. A basic kind of side-effect is the encrypted contract state update produced periodically by `pRuntime`. With the full contract state, a new miner doesn't need to sync the data by replaying the entire blockchain history. Another kind of side-effects is outgoing messages targeting other contracts or blockchains, namely **Egressing Events**. The ability to post messages to external entities is the building block for contract interoperability. The submitted events are eventually dispatched to the target contract, or other blockchains by the Polkadot ICMP (Interchain Message Passing, https://research.web3.foundation/en/latest/polkadot/ICMP/). Miners (host) are responsible to post the side-effects back to the blockchain.

To invoke a contract the user needs to generate a shared secret key with the miner who runs the contract. This can be done via a non-interactive Diffie-Hellman key exchange scheme with her private key and the miner's registered public key [12]. The key is used for future communication and then the user can submit the encrypted payload to the blockchain. The invocation events are processed by `pRuntime` once they arrive at the miner node.

As invocation payloads are included in a block, the blockchain is naturally a canonical source of events. All the contract invocations initiated by users, smart contracts, and other blockchains are timestamped and treated equally by the executor. It therefore makes a unified interface for contract interoperability.

The downside of the architecture is that the confirmation of the commands happens after the confirmation of the block. The performance of the blockchain becomes the bottleneck for contract invocations. However, the read-only queries are made into the runtime directly and the performance is not bounded by the blockchain. This is possible because the queries don't modify the contract state.

Miners are responsible to ensure the communication between `pRuntime` and the blockchain. A monitoring scheme is needed to ensure the connectivity. In the worst case (e.g. miner shutdown) the contract execution can be resumed by another miner.

4.9 State Recover

A single miner is sufficient to run a contract. Though miners are incentivized to run contracts in the long term, the miner may still becomes unresponsive due to network or power outage rarely. In such a case another miner can recover the saved state from the blockchain and resume execution.

As mentioned in section “Execute the Contract”, one of the side-effects produced by `pRuntime` is the periodical contract state update. The dumped state is encrypted by k_{c_i} and stored on the blockchain. In the case of miner unavailable, a new miner can recover the latest dumped contract state from the blockchain and decrypt it with k_{c_i}. After recovering the state, the runtime can further replay the rest of the events on the blockchain until reaches the chain tip.

4.10 Responsiveness Monitoring

Both Gatekeepers and miners are required to keep the responsiveness to keep the functionality of Phala Network. Gatekeepers have to maintain a high level of responsiveness because the root key is kept inside the Gatekeepers runtime and must be available at any time. As long as Gatekeepers can serve the contract key for miners, the availability of the contracts can be guaranteed. So unresponsive miners are not as harmful as unresponsive Gatekeepers to the system.

We adopt a Polkadot-like unresponsiveness detection algorithm [7]. Both Gatekeepers and miners produce side-effects by their runtime. They have to at least post the state updates periodically to the blockchain within an interval. So all the submitted side-effects can be used as a counter of the online activities. Then we can determine if a node n is unresponsive in an era by

$$c_n < \frac{1}{4} \max_{n'} c_{n'}$$

where c_n is the activity counter for node n and n' is all the connected worker nodes in the era. For the detailed design of slash, please refer to Economic Design Paper.

5 Open Question

Security improvements:

- Alternative TEE hardware: Though we use Intel SGX as the reference for the current design, we don’t make any assumptions about the hardware. Potential TEE hardware includes AMD SEV, ARM Turstzone, and some open-source implementations in progress. When we support any alternatives the different TEE can work together transparently.
- Threshold key sharing on Gatekeeper root keys: The confidentiality of all the confidential contracts is derived from the root key. However root keys are replicated among all the Gatekeepers in current design, meaning a compromised TEE could compromise the whole system. A more robust ways is to

apply a threshold secret sharing scheme (e.g. Shamir's) on the root key. A distributed key generation scheme (DKG) can be used instead of selecting a leader to generate the key in the key rotation protocol.

- Contract key backup by secret sharing scheme: To avoid catastrophes where Intel SGX breaks entirely (e.g. Intel bans all the Remote Attestation request from our side), we can utilize a secret sharing scheme to distribute the Root Key to the Gatekeepers, or maybe two generations of the Gatekeepers. In such a case, we can wait for the deployment of an alternative TEE system. Then the secret holders can collaborate to ingest the key to recover the execution of Phala Network.

Optimizations:

- Redundant miners for a contract: Currently one contract is only associated with one miner. The state recovery still takes some time though the availability is guaranteed. With ¿1 miners, as the contract execution is deterministic, the miners can compete to run the contract and checkpoint the states on the blockchain. The blockchain can always accept new states and reject redundant submissions.
- State storage prune
- Layer 2 state sharing

6 The First Application: pLIBRA

pLIBRA is a free and privacy-preserving Libra Coin built on Polkadot. It includes

- A bridge connecting Polkadot and Libra
- A privacy-preserving token pLIBRA 1:1 pegged to Libra Coin (KYC not needed)
- A confidential-preserving cross-chain interoperable DApp ecosystem for Libra users

6.1 System Design

pLIBRA is implemented as a confidential contract on Phala Network. It implements a Libra light client and a ERC20-like token inside the confidential contract.

- **Libra Light Client**: A full light client which is capable to validate the Merkle proof of the chain state and sign transactions.
- **Key Store**: Stores the private keys for the collateral Libra accounts (i.e. *bank account*).
- **AccountDB**: Stores the pLIBRA accounts and the balance.

Confidential contract allows the client to manage privkeys and sign transactions without utilizing computing-intensive cryptographic. Libra users can send their Libra Coin to pLIBRA's bank account and get 1:1 pLIBRA token and vice versa.

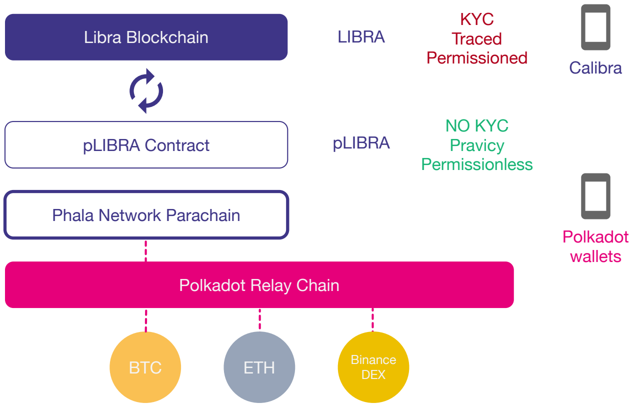

Figure 5: pLIBRA overview.

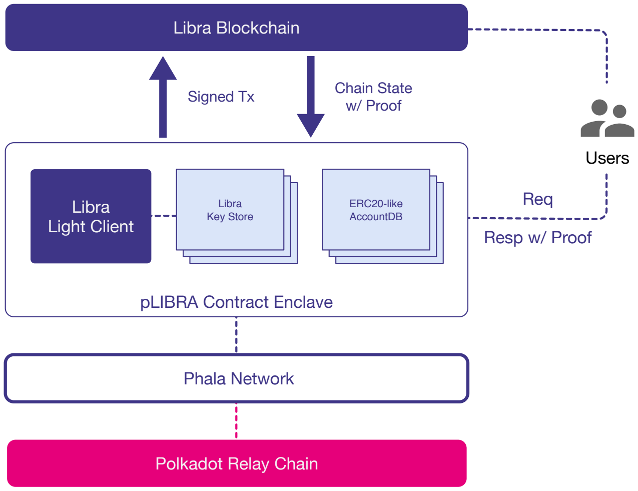

Figure 6: pLIBRA Architecture.

As the information of the pLIBRA accounts is hidden inside the enclave, the pLIBRA token gets privacy-preserving feature like Monero for free. Phala Network is designed to be launched as a parachain on Polkadot. So pLIBRA also gets the ability to interoperate with all the blockchains connected to Polkadot.

At the time of writing, Libra only has very limited smart contract functionality (only whitelisted contracts are allowed). So here we present a naive way to store Libra without the need to employ an advanced feature like multisig or smart contract. When the smart contract is launched on Libra, we can further improve the security of our design by adopting a multisig-style collateral.

- **Init**. During the init phase, pLIBRA contract generates a key pair and reveal the public key. Then an account on Libra blockchain serving as the **Bank Account** is registered with the public key. This is a one-off task done by pLIBRA team. Note that pLIBRA team doesn't have any control over the bank account because the private key is always kept inside the confidential contract.
- **Sync**. The Libra light client is responsible to parse and validate the blockchain data from Libra network. Though at the beginning pLIBRA team will proxy Libra blockchain data to pLIBRA, the blockchain data can be fed by any user. We expect that the pLIBRA end-users will proxy the Libra blockchain data to get their transaction executed. This can be done by a shared library in the wallets. (Side notes: Libra doesn't have a typical "blockchain" data structure. By saying blockchain data, we refer to the Libra ledger updates.)
- **Link Account**. To deposit Libra to the system for pLIBRA, linking the accounts is needed. The user links the address of his Libra account and a pLIBRA account with the signature signed by the key pair of both accounts. It proves that both accounts are owned by the user. After the link operation, deposit and withdraw can only happen between the linked accounts.
- **Deposit**. The user sends some Libra coins to the **Bank Account** and tells pLIBRA the transaction. Once it's confirmed by the Libra light client inside the contract, the same amount of pLIBRA will be issued to the linked pLIBRA account.
- **Withdraw**. The user can request to withdraw pLIBRA. The requested amount of pLIBRA is burned and it will produce a transaction to release the same amount of Libra to the linked address. To broadcast the transaction to Libra network, the pLIBRA contract maintains a sequence of the tx. When the withdraw requester asks for his withdraw tx, it returns all the "not yet executed" tx. The requester has to broadcast all the pending tx until his tx are executed.
- **Transfer**. Transferring pLIBRA token is an internal change in AccountDB. Users can transfer pLIBRA freely like any ERC-20 token. The transfer happens inside the pLIBRA contract and only the owner can access the transaction information.
- **Query**. Users can query the balance and the transaction history of his account. The query operation is a read of Chainview and thus it's not needed to interact with the blockchain.

References

[1] Amd secure encrypted virtualization (sev). https://developer.amd.com/sev/.

[2] Arm trustzone technology. https://developer.arm.com/ip-products/security-ip/trustzone.

[3] Command and query responsibility segregation (cqrs) pattern. https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs.

[4] Event sourcing design pattern. https://docs.microsoft.com/en-us/azure/architecture/patterns/event-sourcing.

[5] Intel sgx: Remote attestation. https://software.intel.com/en-us/sgx/attestation-services.

[6] Intel software guard extensions. https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html, 2019.

[7] Polkadot slashing mechanisms. https://research.web3.foundation/en/latest/polkadot/slashing/amounts/, 2019.

[8] Trusted execution environment. https://en.wikipedia.org/wiki/Trusted_execution_environment, 2019.

[9] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 185–200. IEEE, 2019.

[10] Wood Gavin. Polkadot: Vision for a heterogeneous multi-chain framework. 2016.

[11] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE, 2014.

[12] Philipp Schindler, Aljosha Judmayer, Nicholas Stifter, and Edgar Weippl. Distributed key generation with ethereum smart contracts.

[13] Nicolas Van Saberhagen. Cryptonote v 2.0. https://cryptonote.org/whitepaper.pdf, 2013.

A pRuntime

B Gatekeeper Election

(WIP)

C Network Parameters

(WIP)