Yuan R, Xia YB, Chen HB *et al.* ShadowEth: Private smart contract on public blockchain. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 33(3): 542–556 May 2018. DOI 10.1007/s11390-018-1839-y

ShadowEth: Private Smart Contract on Public Blockchain

Rui Yuan[1], *Student Member*, *CCF*, Yu-Bin Xia[1,*], *Senior Member*, *CCF*, *Member*, *ACM*, *IEEE*
Hai-Bo Chen[1], *Distinguished Member*, *CCF*, *Senior Member*, *ACM*, *IEEE*
Bin-Yu Zang[1], *Distinguished Member*, *CCF*, *Member*, *ACM*, *IEEE*, and Jan Xie[2]

[1] *Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai 200240, China*
[2] *Cryptape Inc., Hangzhou 310007, China*

E-mail: {sjtu_yuanrui, xiayubin, haibochen, byzang}@sjtu.edu.cn; jan@cryptape.com

Received November 14, 2017; revised March 25, 2018.

Abstract Blockchain is becoming popular as a distributed and reliable ledger which allows distrustful parties to transact safely without trusting third parties. Emerging blockchain systems like Ethereum support smart contracts where miners can run arbitrary user-defined programs. However, one of the biggest concerns about the blockchain and the smart contract is privacy, since all the transactions on the chain are exposed to the public. In this paper, we present ShadowEth, a system that leverages hardware enclave to ensure the confidentiality of smart contracts while keeping the integrity and availability based on existing public blockchains like Ethereum. ShadowEth establishes a confidential and secure platform protected by trusted execution environment (TEE) off the public blockchain for the execution and storage of private contracts. It only puts the process of verification on the blockchain. We provide a design of our system including a protocol of the cryptographic communication and verification and show the applicability and feasibility of ShadowEth by various case studies. We implement a prototype using the Intel SGX on the Ethereum network and analyze the security and availability of the system.

Keywords blockchain, smart contract, privacy, trusted execution environment, hardware-enclave

1 Introduction

Blockchain, proposed as an underlying technology of cryptocurrency like Bitcoin, allows users to transfer currency over a distributed, public and trust-less network. Over the last few years, blockchain systems have evolved to support smart contracts which can run custom Turing-complete code on the blockchain, such as Ethereum. Today, public cryptocurrencies are widely used. On Ethereum, more than 10 million ethers① are hold by more than 1 million smart contracts. On these blockchain systems, all the participants have the entire log of the system and reach a distributed consensus on the transactions that will modify the state of the chain. This high degree of replication and the strict consensus mechanism ensure integrity and availability but make all data public, which brings the deficiency in confidentiality.

Previous researchers have proposed several solutions to improve the privacy of blockchain. Bitcoin provides a simple pseudonym-based anonymity to protect secrets, but it exposes all the transactions plainly, which is vulnerable under the attack of relationship analysis[1-2]. Some privacy-preserving cryptocurrencies such as Monero②, Zcash and several others[3-4] do improve the confidentiality of currency transfer, but forgo programmability and cannot support smart contracts. Hawk[5] tries to protect the privacy of both currency transfer and execution of smart contracts. It designs a

Regular Paper

Special Section on Blockchain and Cryptocurrency Systems

This work was supported by the National Key Research and Development Program of China under Grant No. 2016YFB1000104, the National Natural Science Foundation of China under Grant Nos. 61572314 and 61525204, and the Young Scientists Fund of the National Natural Science Foundation of China under Grant No. 61303011.

*Corresponding Author

①Etherscan. https://etherscan.io/accounts/c, Nov. 2017.

②Monero. https://getmonero.org, Nov. 2017.

new coin that is similar to Zcash and requires users to use this coin for private currency transaction. Recently, Microsoft presents an open-source blockchain framework named Coco[③]. Coco enables the creation of a trusted network of physical nodes which is protected by trusted execution environment (TEE). It applies to building a private blockchain network, aka. permissioned blockchain, and can restrict that only the legal members can access the information of the blockchain.

We find that few of these proposed systems can be deployed directly on current widely-used blockchain systems like Ethereum. They either require users to use a new coin (e.g., Hawk), or do not support smart contract (e.g., Zcash). A natural question is: is it possible to support private smart contract on Ethereum?

One of our observations is: in many cases, the privacy of the execution of smart contracts is much more important than the privacy of the entire blockchain. For example, in a second-price auction where the winner pays the second high price, it is critical to hide all the bids during the auction. When the auction is done, the currency transfer information (in the log of blockchain) will eventually be open to the public (e.g., if users use ether to bid). Similarly, in a vote, the most important secret to protect is "who votes whom". Once the vote is done, the result could get public on the blockchain.

Based on the observation, we decouple the protection of the privacy of smart contract execution from the protection of the privacy of the entire blockchain, and further propose a system that can ensure the privacy of smart contract execution. There are several challenges in designing our system. It needs to make a clear separation between the public chain and the private smart contracts, and to define a protocol between the two parts. The system should let any worker node discover new deployed private contracts from the public blockchain, execute them in a protected way, and make the settlement after the execution. The worker nodes are not trusted in that they may leak or tamper with the execution states, or even abort the execution in a malicious way. It is also required to integrate our system seamlessly with existing public blockchain systems like Ethereum without any modification. Finally, for a user, using our private contract should be as easy as using ordinary smart contract.

In this paper, we present ShadowEth, a system that enables private smart contract based on public blockchains. Our idea is to combine hardware enclaves and public blockchains to offer confidentiality of smart contracts while keeping the integrity and availability. On the public blockchain, we create a public smart contract named "bounty contract" which performs the process of deployment and verification and stores the metadata of private contract. We also introduce an off-chain distributed storage named TEE-DS to store binary and states of private contracts. The entire TEE-DS is protected by hardware enclaves and thus all the data it stores will not be leaked or tampered. Users can then publish the deployment and invocation request and the remuneration on the bounty contract to draw workers (who provide off-chain execution environment) in. If a worker wants to execute a private contract, it needs to run a worker client in a hardware enclave, which will get the binary and the state from TEE-DS to execute. After the off-chain execution, the enclave will generate a particular signature and put it back to the Ethereum, which is used to verify the correctness of the execution. Since we just put the metadata (like hash of binary, public key, state versions) and the encrypted data (like input and output) of ShadowEth to the bounty contract, there is no need for any modification to the underlying protocol of existing blockchain systems. Meanwhile, many workers comprise a distributed storage to improve the reliability and guarantee the availability. We also implement a prototype of ShadowEth with the Intel SGX on Ethereum blockchain network and show the applicability with three use cases.

In summary, our paper makes the following contributions.

• It presents ShadowEth, a confidential, distributed, trust-less off-chain smart contract system clinging to existing public blockchain networks like Ethereum without any modification.

• It describes the detailed architecture and protocol of ShadowEth.

• It shows the applicability of ShadowEth with three use cases.

• It presents a prototype and demonstrates the security and availability of ShadowEth.

The rest of the paper is organized as follows. We present the motivation of this paper and previous technologies in Section 2. The high-level architecture of ShadowEth is introduced in Section 3. The detailed design of ShadowEth is demonstrated in Section 4. Three cases are presented to show the applicability of ShadowEth in Section 5. A prototype is presented and the security of ShadowEth is analyzed in Section 6. The

[③]Microsoft Corporation. Coco-framework. https://github.com/Azure/coco-framework, Nov. 2017.

availability of ShadowEth is discussed in Section 7. Related work is presented in Section 8. Finally this paper is concluded in Section 9.

2 Background and Motivation

In this section, we provide background on the technologies that underpin ShadowEth. We first give a short overview of the blockchain and smart contracts, explore the lack of confidentiality of current smart contract systems, then introduce the hardware enclave, and finally describe the threat model of ShadowEth.

2.1 Blockchain

A blockchain typically serves as an open, decentralized and trustless distributed ledger which is maintained by all participants. Some participants, called miners, form a peer-to-peer network and all have the full copy of the blockchain. They collect transactions signed by users. After validating the signatures, they packed these transactions into a block. Each block contains the information of the transactions as well as the hash of the previous block, which organizes the data as a sequential list of blocks, called a blockchain. The blockchain is a distributed system designed for Byzantine fault tolerance. Each transaction which may modify the state of the chain will be broadcasted to all miners in the network. Once a block is generated, all the miners need to achieve a consensus on whether to accept it or not. Each miner can decide the block's content arbitrarily, that is to say, he/she can decide which transactions will be packed into the block. Miners can always generate different blocks with the same parent at the same time, which will cause inconsistency called a fork. To solve the bifurcation, an honest miner always chooses to follow the longest branch.

If an attacker controls more than 50% of the nodes, he/she can unilaterally generate the branch containing the fake transactions faster than the branch containing the real ones, which causes the double spending problem. To solve this problem, a miner needs to prove that he/she has done a certain amount of work before generating a block. With this proof, known as Proof of Work (PoW), the block is acknowledged to be valid. PoW makes a miner create blocks at the rate related to the proportion of his/her mining power, which prevents Sybil attacks.

2.2 Smart Contract and Ethereum

The smart contract can trace back to 1996, proposed by Szabo[④]. It is described as "a set of promises, specified in digital form, including protocols within which the parties perform on these promises". The smart contract is usually designed to ensure the execution of a contract and to avoid malicious actions as well as unforeseen circumstances. It can minimize the utilization of the trusted third party, which results in the reduction of transaction cost, and potentially circumvent censorship, collusion and counter-party risk.

Blockchain makes smart contracts possible. Based on the blockchain, smart contracts usually take the form of a general-purpose program. Users can write and deploy any Turing-complete program on the blockchain network. The most notable smart contract implementation is Ethereum[⑤]. A contract in Ethereum will be endowed with execution contexts such as stack, heap and persistent memory on the chain. Once a contract is deployed, it will be executed autonomously. Even its creator cannot stop the execution or modify the code. A contract can operate as a specified function, accept messages as arguments, and eventually update its state. The execution of a contract is triggered by a message from a user account or another contract, analogous to a function call, and is finished when the program exits or the gas (the fee paid for miners) is depleted.

Ethereum provides a runtime environment named Ethereum Virtual Machine (EVM). It is sandboxed and isolated from the host operating system. Each miner runs an EVM for contract execution. In Ethereum, the smart contract is a high-level programming abstraction. Smart contracts can be written in a programming language like Solidity. Then the source code will be compiled to bytecode for EVM and deployed to the Ethereum network in a transaction. Ethereum has its value token called ether[⑥]. Analogous to Bitcoin, it is a cryptocurrency with its market value. The settlement of smart contracts is done on the base of ether. To prevent DoS attacks such as requests for executing some infinite loop within smart contracts, contracts need to be powered by a certain amount of ether called gas.

④Szabo N. Smart contracts: Building blocks for digital markets. 1996. http://www.alamut.com/subj/economics/nick_szabo-/smartContracts.html, Mar. 2018.

⑤Buterin V. Ethereum: A next-generation smart contract and decentralized application platform. 2014. https://github.com/ethereum/wiki/wiki/%5BEnglish%5D-White-Paper, Mar. 2018.

⑥https://coinmarketcap.com/currencies/ethereum/, Nov. 2017.

Ethereum endows every operation, including computation and data transfer, with a fixed price, and the corresponding gas will be consumed once the operation is executed. A transaction must contain a parameter named gas limit, which defines the maximum limit of the gas consumed by the execution. Once a contract calls another function, it needs to specify a lower gas limit for the function. If a function exits normally, it will consume the corresponding gas and return the rest gas. When a function runs out of its gas, it will be aborted and all the changes of the states caused by it will be rolled back to their pre-call states without returning any gas.

The wide public participation of Ethereum and the strict consensus mechanism ensure the enforcement, integrity and availability of smart contracts. However, the lack of privacy becomes a pain point that restricts the applications of smart contracts in some privacy-sensitive scenarios.

2.3 Hardware Enclave

Trusted executed environment (TEE) is a new feature provided by recent commodity CPUs. It creates a secure area which guarantees the integrity and confidentiality of code and data inside. TEE serves as an isolated environment running in parallel with OS. It provides a high-level security for software inside by reducing the trusted computing base (TCB) to only CPU. Applications running in TEE have secure memory and cryptographic operations to resist attacks from other applications, even the privileged software such as OS or hypervisor.

Our design is general-purpose and applies to any TEE that has above features. In this paper, our implementation is based on Intel's Software Guard Extensions (SGX)[6]⑦⑧. Intel SGX provides a trusted and isolated environment called enclave. With this hardware feature in CPU, users can deploy their softwares in a remote host with integrity and confidentiality unless CPU package was hacked. An application running inside an enclave is protected from other malicious software including the operating system.

SGX provides remote attestation⑨ which allows a remote host to verify the application running in the enclave and generate a secure channel to communicate with it. In the process of initiation of an enclave, the CPU measures the trusted code and the trusted memory within the enclave and produces a hash based on all memory pages known as measurement. Then the software inside the enclave can acquire a report which contains the measurement and other supplementary data such as the public key. This report is signed by a hardware-protected key in CPU to prove that the measured software is running in SGX indeed. The remote attester can then verify the report with Intel Attestation Service (IAS) which can certify that the signature is valid and the corresponding report is generated from authentic CPUs.

2.4 Threat Model

Our threat model assumes that multiple parties mutually distrust each other. They are potentially malicious, and may try to steal information of smart contracts, modify the execution flow, and deviate from the protocol for their benefit. Each party may send, drop, modify, and record arbitrary messages in the protocol at any time during the contract deployment and invocation. Any party may crash and stop responding entirely.

We assume that the blockchain is trustable and available all the time. The information on the blockchain is tamper-resistant but public to everyone. We also assume that network adversaries can intercept the communication between parties, but they cannot control the whole network so that the communication can be eventually established, for example, the user can send a request to the blockchain network and get the response.

We trust the hardware enclave, its manufacture (like Intel) and the remote attestation service. As long as a node passes the remote attestation, it will be able to execute the shadow contract within its enclave. The rest of the system, including the other software stacks (outside the enclave) and the hardware, is not trusted. Side-channel attacks[7-10] against enclaves and DoS attacks are not considered in this paper.

Our system also relies on the privacy of private key. Each private contract has a unique private key only

⑦Intel Corp. Software guard extensions programming reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, Mar. 2018.

⑧Intel Corp. Intel software guard extensions SDK. https://software.intel.com/en-us/sgx-sdk, Mar. 2018.

⑨Johnson S, Scarlate V, Roza C *et al.* Intel software guard extentions: EPID provisioning and attestion services. https://software.intel.com/sites/default/files/managed/ac/40/2016%20WW10%20sgx%20provisioning%20and%20attesatation%20final.pdf, Mar. 2018.

possessed by the enclave. The private key is used to generate the attestation that the contract has been executed correctly. If an attacker steals the private key in some way, he/she can get paid from the Ethereum without executing the contract, which could compromise the integrity but not the privacy.

3 System Overview

The goal of ShadowEth is to provide a confidential platform to execute private smart contracts which can be integrated with existing public blockchain such as Ethereum. Specifically, the privacy of a smart contract consists of the following three parts.

• *Privacy of the Specification of a Smart Contract.* The source code of a private contract must be hidden during the deployment and subsequent process of execution and synchronization.

• *Privacy of the Execution of a Smart Contract.* Once a private contract is invoked, the executing process on a worker client cannot be spied and the call arguments as well as the return values should be hidden during the execution.

• *Privacy of the State of a Smart Contract.* The internal state of a private contract may contain users' secrets and can reflect the information of recent transactions. Therefore it should not be published on the blockchain.

To guarantee the confidentiality of the code and data of a smart contract, a secure channel between a user and TEE-DS will be established before transferring the contract. The contract will be encrypted before transferring and can only be decrypted inside the corresponding enclave.

To preserve the privacy of execution, we only put the information of invocation and verification onto the blockchain. During the deployment of a private contract, TEE will generate a key-pair for this contract and publish the public key. The invocation arguments are encrypted with the contract's public key which can only be decrypted within the enclave. The return value will be encrypted by a user-provided key which is delivered along with call arguments. In the entire process, the information of the execution is encrypted except inside the enclave. Anyone even the worker cannot leak the internal executing state.

To guarantee the confidentiality of the persistent state of a private contract, ShadowEth stores only the hash of the ledger on the Ethereum instead of all data. The data can only be managed and viewed inside enclaves. Due to the limited secure memory of enclave like SGX, data could be moved to untrusted memory or disk, and sometimes need network transmission for backup or synchronization. Before writing out the data, ShadowEth will encrypt all data with the hardware key which is only kept by CPUs. This ensures that the data can only be accessed by authenticated users through ShadowEth and no one outside can view or manipulate the persistent state.

3.1 System Components

Fig.1 shows the architecture of ShadowEth. ShadowEth can be broken down into several subcomponents.

• *Bounty Contract.* A bounty contract is a native

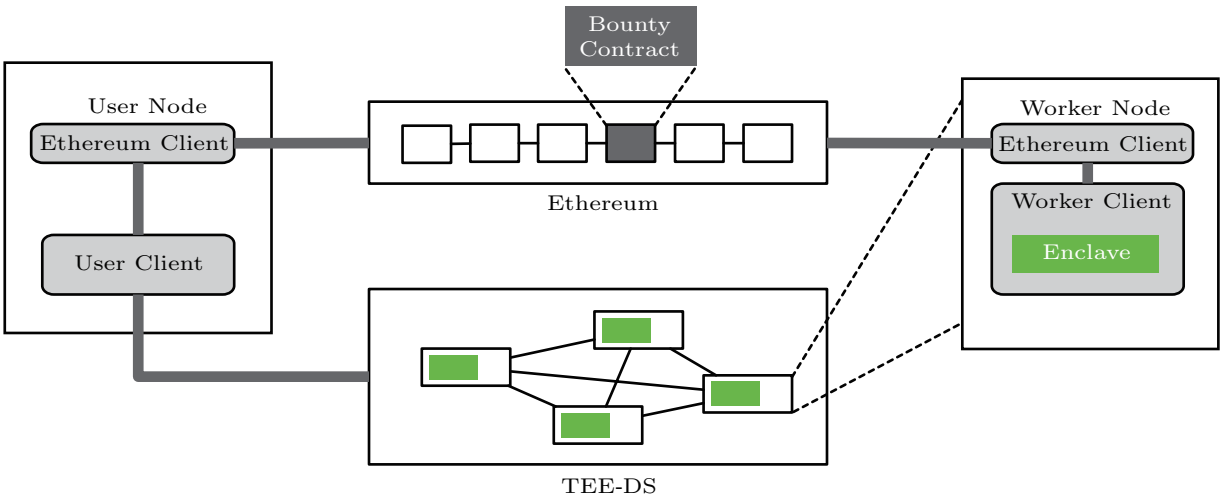

Fig.1. ShadowEth architecture.

smart contract deployed on blockchain directly, serving as the public portion of ShadowEth.

• *User Client.* A user client provides interfaces to end users. It does not require any enclave to execute.

• *Worker Client.* A worker client is responsible for the execution and maintenance of private contracts. It needs to run inside enclaves.

• *TEE-DS.* TEE-DS (distributed storage) serves as a distributed network which stores private contracts.

We create a smart contract named bounty contract on the Ethereum blockchain network which serves as a platform for publishing private contract, grabbing execution task and remuneration settlement. Bounty contract is a public contract that every participant in Ethereum can view and use. A user can deploy his/her private contracts and then invoke them like ordinary contracts via the bounty contract. On the other side, workers with TEE-enabled devices can look up executory tasks in the bounty contract, get the parameters, do the computation, and finally commit the result of the execution.

A user client provides interfaces including contract deployment and invocation for users. It communicates with the bounty contract through an Ethereum client. It can be launched without TEE, thereby it is only trusted by its user.

A worker client is used for fetching executory tasks from bounty contract, getting code and data of private contracts from TEE-DS, executing contracts, committing results, and updating the persistent state of contracts. The main part of a worker client, which does the contract correlation processing, runs inside enclaves. It communicates with bounty contract through an Ethereum client. Furthermore, a worker client also serves as a server node in TEE-DS.

Many worker clients comprise TEE-DS, a peer-to-peer network storing the code and data of private contracts. In the process of execution of contracts, all the nodes maintain consistency by Paxos-like consensus algorithm. For a certain contract, different workers store the same data but encrypt it with different secret keys. Enclaves maintain the consistency of the unencrypted logical data. The data synchronization is performed on the safe channel between two enclaves after the remote attestation.

3.2 Example

Here is a simple example to briefly demonstrate the process of deployment and invocation from different perspectives in our system.

Deployment. Suppose a user needs to deploy a private contract to ShadowEth. He/she first compiles the code and puts the binary to TEE-DS through the user client. TEE-DS will generate a pair of keys, bind them with the contract, and transfer only the public key back to the user. The user client then uploads identification information of the contract (including the public key and the hash of binary, etc.) to the bounty contract. Now the contract is publicly available.

Invocation. Once the user needs to invoke his/her private contract, he/she sends an invocation request (including arguments) to the bounty contract with a sum of remuneration. A worker client will get the arguments from the bounty contract and get the private contract's binary from TEE-DS with the public key as its ID. It then loads the binary to its hardware enclave and executes using the arguments to get a return value. After that, it sends the new state of the private contract to TEE-DS, which will be held at this moment. The worker then makes a response (including the return value and signature, etc.) and sends it to the bounty contract. The bounty contract will verify the response to ensure that it is generated by correct contract, states, and arguments in real hardware enclave before transferring the remuneration to the worker. Once TEE-DS confirms that the execution has been acknowledged by the bounty contract, it will update the states of the private contract, which finishes the process of one invocation.

As shown in the example, the Ethereum ensures the availability (a private contract will eventually get executed), and the integrity (the result cannot be modified), while the hardware enclave is used to protect privacy. There are many challenges unlisted, e.g., how to design a protocol and key management to defend against attacks like rollback and impersonation, how to minimize the trust on the manager in a second-price auction, which will be described in Section 4.

4 Design

In this section we present the design of ShadowEth. We first describe the three major parts of ShadowEth: the bounty contract (Subsection 4.1), the shadow contract (Subsection 4.2) and the TEE-DS (Subsection 4.3), and then introduce the detailed protocol (Subsection 4.4).

4.1 Bounty Contract

Bounty contract is a native smart contract deployed on Ethereum. Its major responsibility is to perform the

public portion of deployment, invocation and verification of private contracts. Sometimes it needs to generate transactions to handle the settlement. It maintains two lists: a contract list and a todolist.

Each entry in the contract list represents a private contract. It contains 1) the contract's ID (as the primary key), 2) the contract's public key, 3) a version number, 4) an owner list, 5) the hash of the contract's persistent state, and 6) the balance of the contract. Once the bounty contract receives a deploying request, it will construct a new entry and add it into the contract list, and the invocation will update the hash value and increase the version number. The contract list can be used to record and verify the state of private contracts but does not expose any information about the core business logic. Users can deposit funds into a private contract by sending the corresponding ether to bounty contract through deploy transaction or invoke transaction (described in Subsection 4.4). The bounty contract records the funds as the balance of a contract. When the result of an invocation is a settlement, the bounty contract will check if the balance is enough before performing the transfer. It is worth noting that the bounty contract only records the total balance of the contract. The detailed distribution of this sum money is decided by the contract itself.

The todolist serves as a task pool. Users publish invocation tasks into the todolist to allow workers to bid the task. Each entry represents an invocation request, which contains 1) the task ID (as the primary key), 2) the contract's public key, 3) the encrypted arguments, 4) the remuneration offered by the user, 5) the state (e.g., TODO or FINISHED), and 6) the encrypted return value. Only one worker can gain the remuneration (typically the first), which is guaranteed by the Ethereum.

The bounty contract is the key component of our system. Users and workers communicate indirectly through the bounty contract. Thus, the integrity and the availability of operations on private contracts like deployment and invocation are ensured by the public blockchain.

4.2 Shadow Contract

We propose to build a confidential environment for smart contract execution using hardware enclave, but it is not enough to just deploy the native Ethereum contract directly because only the process of the execution can be hidden, while the information outside such as the call arguments and return values is still exposed.

To this end, we introduce Shadow Contract, a redesign of the native Ethereum smart contract. Shadow Contract uses a contract gate to isolate the core business logic of the smart contract. The contract gate is loaded by the worker client before the execution of smart contracts, endowed with the private key of the smart contract. The contract gate has two primary functionalities: decrypting arguments and generating the response. The two functionalities are independent of the contract code, which means the contract gate can be compatible with any private contract.

To preserve the privacy of the call arguments, users need to encrypt them with the public key of the contract before sending them to the bounty contract. Before execution, the contract gate first decrypts the arguments and then invokes the target function with the plaintext.

After the execution, a response is required to put back to the bounty contract. A response includes the execution's return value, the version number before the execution, the hash of the contract's state after the execution, and the settlement information. What is more, an invocation verification signature (IVS) is required to be attached to the response.

IVS is used for verifying whether the contract has been executed correctly inside an enclave. The contract gate signs the response along with the hash of call arguments and the worker's ID by the secret key of the contract to generate IVS. The encryption can only be performed after the execution within enclaves. Then the bounty contract can decrypt IVS by the public key of the contract to verify the execution. The inclusion of the hash of call arguments is to ensure that the worker does execute the contract with given arguments. Since IVS contains the ID of the worker, the remuneration will be sent to the right worker even when a malicious attacker captures IVS and resents it to the bounty contract using his/her own Ethereum account.

The return value of the execution is required to be sent back to the user without exposure. To this end, the user needs to provide another symmetric key along with the arguments and encrypt them together with the public key of the contract. The contract gate will encrypt the return value with this key and put the cipher-text into the response, which will then be put back to the bounty contract, and only the user can decrypt it.

When a function is to trigger a settlement on the Ethereum, it will return a transaction-type object which the contract gate will put into the response. If a response contains settlement information,

the bounty contract will generate an Ethereum transaction to transfer the money.

Besides the above-mentioned data, the contract gate also puts the hash of the contract state after execution into the response which is used as an attestation for off-chain contract transfer.

4.3 TEE-DS

TEE-DS is a peer-to-peer network consisting of many worker clients. It serves as a distributed storage of private contracts which can provide high reliability against malfunction. The secrets (e.g., the code, data and private key) of a private contract are protected by hardware enclave, which guarantees the confidentiality.

Admission Mechanism. Anyone who runs the worker client inside an enclave can join TEE-DS as a worker. To get permission, the expectant worker needs to connect one of the approved workers and offer a CPU-signed statement that he/she is executing a particular enclave. Once the statement is certified, which is known as remote attestation, the expectant worker can then join the network, get the current network constitution, and synchronize data from other workers.

Synchronization Mechanism. After a worker performs an invocation of a private contract, he/she needs to broadcast the updates to other workers. However, some workers may publish different updates of the same contract simultaneously. This happens for several reasons: 1) the states before invocation are different; 2) the invocation requests they choose are different; 3) there are some malicious actions. We utilize the consensus on the blockchain to evade conflicts in TEE-DS. When a worker sends the response of an invocation to the bounty contract, he/she must specify the version number before the execution and the hash of the contract's state after the execution. When receiving more than one response about the same contract, the bounty contract will check if the version number is the latest and only accept the first valid one. Then the bounty contract increases the version number and updates the hash. Even if different miners accept different responses at first, they will eventually reach an agreement by Ethereum's consensus mechanism. In TEE-DS, before the data synchronization, workers will check the version number as well as the hash in the bounty contract and only accept the updates which have been acknowledged.

4.4 Protocol

The ShadowEth protocol operates in two scenarios: 1) contract deployment, and 2) contract invocation. Fig.2 shows the process. Just like smart contract systems based on public blockchains, our current system does not support withdrawal mechanisms unless they have been coded in the contract. Considering the irreversibility and non-repudiation, a contract cannot be stopped from the outside once deployed. The following is a detailed description of the process of these two scenarios. For simplicity, we ignore mining fees in this subsection, although they can be supported in the implementation.

4.4.1 Contract Deployment

The first scenario of the ShadowEth protocol is contract deployment, as shown in Fig.2(a). Similar to using Ethereum natively, users write the business logic of their private contracts on their clients using native languages (like C/C++), then compile the code, deploy them onto TEE-DS, and meanwhile upload the identification information (e.g., the hash of the code) to the bounty contract.

The final purpose of the deployment is to generate an asymmetric key pair, of which the private key is only kept by the enclave, so that the subsequent invocation can be protected by the public key directly without establishing a secure communication channel.

First, the user sends the binary code to TEE-DS through a secure channel, which is established through remote attestation and protected by a Key_{session} which is used for encrypting data in the session. Once the code is received, TEE-DS will generate an asymmetric encryption key pair (based on the RSA algorithm): Key_{c_p} and Key_{c_s} (p for public and s for secret), which is unique for each contract. Then TEE-DS sends Key_{c_p} back to the user.

After receiving Key_{c_p}, the user will upload the identification information to the bounty contract, essentially announcing the existence of a new private contract. The information includes: 1) Key_{c_p}, 2) the owner list (the user's public address as default), and 3) the hash of the binary code. This is done by the Ethereum client, which generates a deploy transaction containing the information and sends it to the bounty contract. The bounty contract then creates a new contract record in its contract list with the identification information, sets the version number to 0, and sets the state to DEPLOYED.

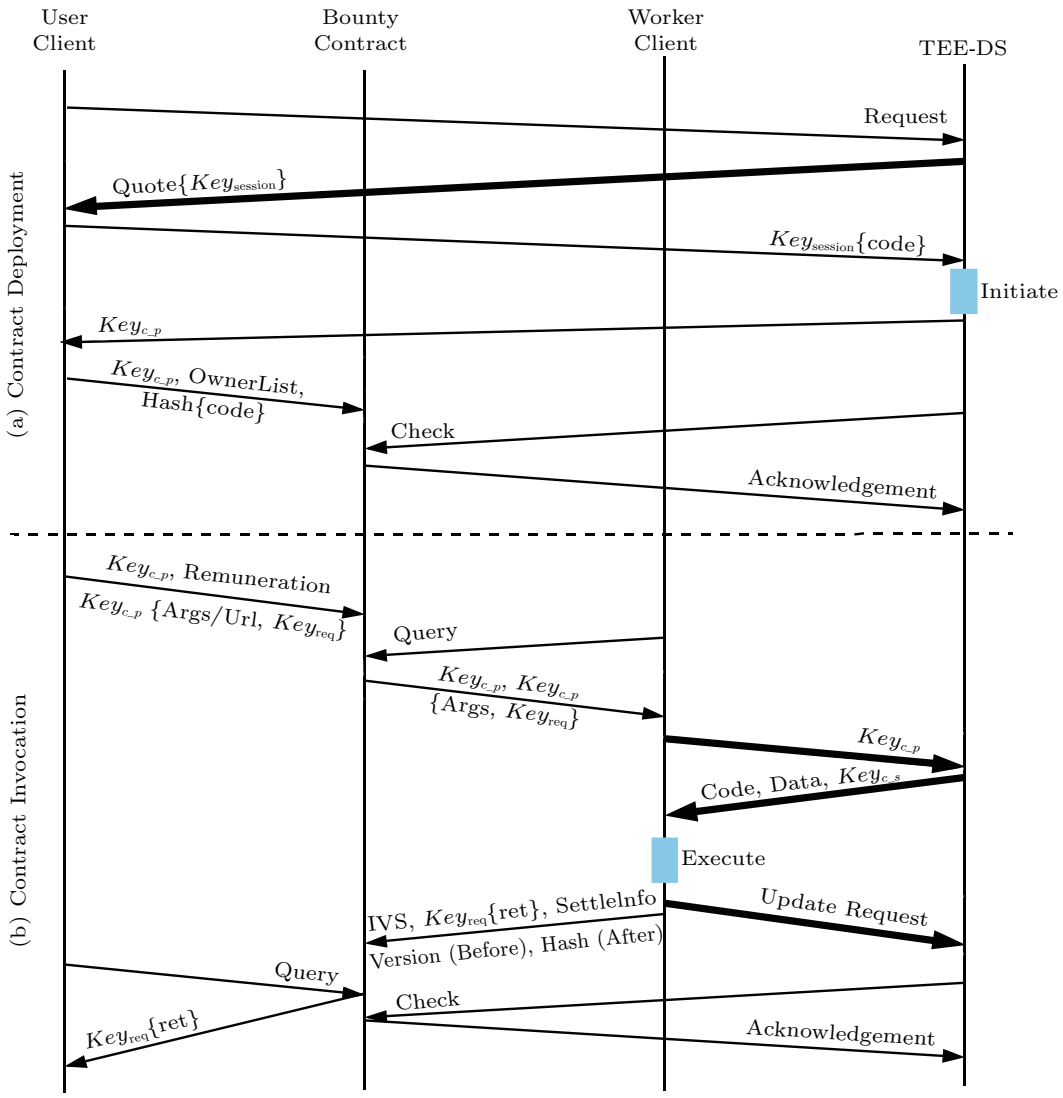

Fig.2. ShadowEth protocol. Bold lines represent secure channels protected by enclave.

At last, after the deploy transaction is acknowledged, the code of the private contract along with the key pair will be broadcasted to all other workers in TEE-DS.

4.4.2 *Contract Invocation*

Once a private contract has been deployed, users can invoke it just like invoking native Ethereum contracts. The procedure is shown in Fig.2(b).

1) A user initiates an invocation request through its user client. Similar to the native contract invocation in Ethereum, the user must specify the contract's Key_{c_p}, the remuneration, and the corresponding parameters including the function name and call arguments.

2) The user client will process the request: a) adding the timestamp into the request body, b) generating a secret key Key_{req} which is only used for this request and adding it into the request body, and c) using Key_{c_p} to encrypt the request except for the remuneration, and then send the data to the Ethereum client.

3) The Ethereum client generates an invoke transaction including the contract's Key_{c_p}, the encrypted request, and the remuneration, which is sent to the bounty contract.

4) Once the bounty contract receives the invoke transaction, it first verifies the identification to ensure that the contract exists and the request is from one of the owners of the contract. It then adds a new entry with the information included in the invoke transaction into the todolist marked as TODO, and transfers the remuneration into its account simultaneously.

5) After the invoke transaction is acknowledged,

workers can see the executory tasks. They can choose any task and get the related information including Key_{c_p} and encrypted request from the bounty contract.

6) By using Key_{c_p}, a worker can ask TEE-DS for the contract's code and load the contract into its enclave along with the encrypted request. Inside enclave, the contract gate first decrypts the request to get the arguments and Key_{req}, and then executes the specified function.

7) If the function exits normally, the contract gate will broadcast the modification of the contract's persistent state to other workers in TEE-DS and then generate a response including the following parts: a) the version number of the contract before the execution, b) the hash of the contract's state after the execution, c) the return value (if the function has one) which is encrypted by Key_{req}, d) the settlement information (if the function has one), and e) the IVS (described in Subsection 4.2). The worker client then sends the response back to the bounty contract, and the bounty contract can verify the validity of the IVS with Key_{c_p} to ensure that this worker has completed this task correctly. After the verification, the bounty contract updates the entry in the todolist, marking it as FINISHED and filling the return value if any, and updates the information of the contract including the version number and the hash of contract state. Then the user can fetch the return value and decrypt it by Key_{req}. If the result is marked as a settlement, the bounty contract will generate a settlement transaction with the settlement information in the response. The first worker to finish it will obtain the remuneration.

8) Once the result is acknowledged by bounty contract, the other workers in TEE-DS will accept the modification.

5 Use Cases

In this section we introduce some use cases to explain how we can use ShadowEth to preserve the privacy of smart contract.

5.1 Protecting Simple Vote

We first implement a simple vote example to show the confidentiality of private contracts. The scenario is that some people want to start a vote between themselves by a smart contract on the Ethereum, but they do not want to expose the content of this vote to the public. In our implementation, we assume that the participants know and trust one another and create the vote contract together. Algorithm 1 shows the approximate logic of the example but not the detail.

Algorithm 1 . Pseudo-Code of Simple Vote Contract

```
1: vote ← {0, 0, 0, 0, 0}
2: function CAST(option)
3:     vote[option] + +
4: function CHECK()
5:     return new ResultType(vote)
```

First, all the voters agree on the voting code (private contract). Then they deploy the contract as described in Subsection 4.4, which can be performed by any one of the participants. After the deployment is acknowledged, each participant can invoke the *cast*() function to cast their ballot. The *cast*() operation will be executed by workers using hardware enclaves. Any participant can invoke the *check* function to query the current state of the vote at any time.

It is noteworthy that the whole process of the vote including the *cast* and the *check* is hidden and only the participants can view it. The workers can get involved in executing but they will never know what indeed happens inside the contract. If a participant is malicious in this case, he/she can only leak the vote result but not the detailed information.

5.2 Protecting Transaction Details

We consider the following scenario: a seller and a buyer have a contract with a certain price and quantity of some commodity. The key secret of the scenario is the detail of the purchase including the price, the quantity and maybe some promotion strategy in some complicated cases, while the result (i.e., the total *ether* transferred) which will eventually be reflected on the Ethereum is not protected. The approximate code is shown in Algorithm 2. To ensure the efficacy of the contract, the seller is required to deposit some funds as balance before making a purchase, which is done by invoking the *deposit*(), and send corresponding *ether* to the bounty contract. Once the both parties are prepared to make a deal, they can invoke *purchase*() with the negotiated price and quantity. The invocation of *purchase*() requires to be signed by both parties. Either party can invoke *settlement*() for a refund. The bounty contract will check the total amount of refund before generating the settlement transaction.

The example can be extended to a multi-participants scenario which can hide the detail of transactions among the participants and do settlement regularly. Although the transfer of ether is public on

Ethereum, it is acceptable in many cases because the privacy of transaction details is much more important. And users can do a settlement after a number of transactions, which mixes up the results of these transactions and makes it difficult to resolve.

Algorithm 2. Pseudo-Code of Private Transaction Contract

```
1: int seller_addr                                          ▷ address
2: int buyer_addr
3: int seller_blc                                           ▷ balance
4: int buyer_blc
5: function DEPOSIT(amount)
6:     buyer_blc+ = amount
7:
8: function PURCHASE(amount, price)
9:     ▷ check the signature first
10:    if buyer_blc >= amount × price then
11:        buyer_blc− = amount × price
12:        seller_blc+ = amount × price
13:        return newResultType(true)
14:    else
15:        ▷ transaction failed
16:        return newResultType(false)
17:
18: function SETTLEMENT()
19:    settlement ← new Transaction()
20:    settlement.add(seller_addr, seller_blc)
21:    settlement.add(buyer_addr, buyer_blc)
22:    return settlement
```

5.3 Second-Price Auction

In a second-price auction, the bidder who offers the highest price wins but pays the second highest price. The essential element of second-price auctions is that bidders offer bids without knowing the bid of other bidders.

We implement an example auction program using ShadowEth, as shown in Algorithm 3. The code above is an approximation of our real implementation. There are two different roles in this case: the manager and the bidders. First the manager can start an auction with the seller's address so that the fund can be transferred to the seller immediately once the auction ends. After the auction starts successfully, each bidder can offer their price by invoking *bid*(). Since the ether transfer on Ethereum is public, from which the bid price could be inferred, we allow a user to obfuscate the real price by sending an arbitrary (but more than the real price) amount of ether to bounty contract, and the excess will be returned to the user after the auction ends. The manager will decide when to conclude the auction and invoke *conclude*() to transfer the money from the winner to the seller and refund other bidders' funds.

ShadowEth guarantees the input independent privacy that each user can never see others' bids even after the auction. In this way, users' bids are independent of others' bids. Also, the manager's function is limited to starting and terminating the auction. Even if the manager is malicious, he/she cannot disclose any information of the auction.

Algorithm 3. Pseudo-Code of Second-Price Contract

```
1: Map⟨int, int⟩ balances
2: int bestPrice ← −1
3: int secondPrice ← −1
4: int winner ← −1
5: int seller ← −1
6: function START(addrOfSeller)
7:     balances.clear()
8:     seller ← addrOfSeller
9:
10: function BID(addr, price, funds)
11:    ▷ check: funds deposited ⩾ real price
12:    balances.insert(addr, funds)
13:    if price > bestPrice then
14:        secondPrice ← bestPrice
15:        bestPrice ← price
16:        winner ← addr
17:    else if price > secondPrice then
18:        secondPrice ← price
19:
20: function CONCLUDE()
21:    settlement ← newTransaction
22:    settlement.add(seller, secondPrice)
23:    settlement.add(winner, balances[winner]
24:    −secondPrice)
25:    for each b ∈ balances do
26:        if b.first! = winner then
27:            settlement.add(b.first, b.second)
28:    return settlement
```

6 Evaluation

We implemented a prototype using Intel SGX on the Ethereum testnet. We demonstrated that ShadowEth achieves security and availability with acceptable overhead in this section.

6.1 Prototype

The prototype contains the three major components: the bounty contract on Ethereum, the user client, and the worker client.

The bounty contract is written in Solidity, a high-level language designed to target the Ethereum Virtual Machine. We implemented the deploy, the invoke, the submit interfaces for off-chain users and workers. The bounty contract holds the funds deposited by all private contracts and can generate Ethereum transactions to redistribute them.

The user client and the worker client are written mainly in C and C++. Both communicate with

Ethereum nodes through JSON-RPC interfaces including *eth_sign*, *eth_sendTransactions* and *eth_call*. For the worker client, we implemented a more complete and usable software stack than the official SDK (software development kit) and ported a memcached instance into an SGX enclave which is used as a distributed storage. We also implemented the contract gate inside the enclave to load and execute contracts. For asymmetric encryption between the outside and the inside of the enclaves, we used RSA with 4 096-bit keys.

In contrast to some performance optimization solutions such as TEEchan[10], each execution in ShadowEth needs the intervention of Ethereum which is the performance bottleneck. Each transaction in Ethereum takes about 12 seconds to be packaged into a block and the time is always extended to about 1 minute for five confirmations. Furthermore, the confirmation time is related to the fee paid to the Ethereum miners. For some simple contracts, the off-chain computation costs several seconds (mainly for the decryption and signature in our current implementation), which is an order of magnitude less than the confirmation time. Therefore we can measure the performance by the times that off-chain components communicate with the on-chain components. Currently, it is required to wait for two transactions to be acknowledged in each execution which is acceptable in most cases.

6.2 Security Analysis

In this subsection, we discuss how ShadowEth mitigates potential attacks. Each party may send, drop, modify, and record arbitrary messages in the protocol at any time during the contract deployment and invocation. Therefore we discuss and evaluate the security of ShadowEth.

Malicious Worker. During the contract deployment, an attacker may pretend to be a worker and defraud the user of the code of his/her private contract. To defend this attack, remote attestation is required before the communication. The worker must provide a report which is signed by the hardware-protected key to prove that he/she runs the unmodified worker client in enclave indeed. Even if the attacker has compromised the network, he/she cannot spy or tamper any message because the communication between the user and the worker is based on a secure channel which is protected by a session key after remote attestation. If the user who performs the deployment is dishonest, he/she cannot get any more privileges than other users in that the secret key of the contract is generated and kept by enclaves.

Stealing Invocation Information. The contract invocation is mediated by bounty contract which is publicly visible on Ethereum. An attacker may try to steal the invocation information by spying the bounty contract. But this will not work because all the secrets of invocation are encrypted. The user encrypts the arguments with the public key of the contract, and oppositely, the enclave will encrypt the corresponding return value by another key which is transferred along with the arguments by the user. Therefore the attacker can only see the inessential information such as the amount of the remuneration. Since the secrets of an invocation are fully hidden except for the invoker, ShadowEth can guarantee the personal privacy without any exposure risk from a malicious manager, which is a potential concern of some other private blockchain system such as Hawk[5].

Integrity of Invocation. There are potential attackers including dishonest workers who want to compromise the integrity of the invocation by tampering invocation requests or committing fake results. Since the invocation requests are acknowledged on Ethereum, attackers can manipulate them only by controlling more than half of the computing power of Ethereum, which is considered impossible. The response of an invocation includes IVS which contains the hash of corresponding arguments and is signed with the contract's private key by the enclave. The bounty contract can check the IVS with the contract's public key to verify that the worker executes the contract correctly with the given arguments inside an enclave. Therefore no one can fake a response to the bounty contract.

Replay Attack. To protect the private contract from the replay attack, each message will be endowed with a timestamp. When the bounty contract receives an invoke message, it will first check the timestamp and refuse the old requests. The communication between users and workers will be protected in the same manner. Furthermore, the response of an invocation must specify the corresponding task in the todolist of the bounty contract. Thus the only one response will be accepted by the bounty contract for one task. It is worth noting that the worker's address inside the IVS decides who will gain the remuneration and thus it is no use for attackers to intercept the IVS.

[10] Lind J, Eyal I, Pietzuch P *et al.* TEEchan: Payment channels using trusted execution environments, 2017. https://arxiv.org/pdf/1612.07766.pdf, Mar. 2018.

We implemented ShadowEth using Intel SGX which guarantees the integrity and the confidentiality of the execution. Intel SGX can protect the execution of smart contracts from attackers on the same host, even those who compromise the OS or control physical access. All the data of the contract is encrypted except inside the enclaves. Malicious workers or attackers on the same machine can only stop service, modify or record the encrypted messages, which will not harm the integrity and confidentiality of our system. For the attacks which exploit the hardware vulnerabilities (e.g., Meltdown and Spectre CPU security flaws), Intel has submitted patches to fix the problems. Actually, most side-channel attacks require multiple attempts to steal information from the enclave, and thus we can prevent these attacks effectively by limiting the execution times. This is a good idea and we will probably implement in the future.

It is worth noting that our protocol is not Intel specific and we can implement our system easily on the base of another trusted hardware.

7 Discussion

Availability. Currently, the availability of a blockchain system depends on the participation to a large extent. Similarly, ShadowEth needs a number of workers to comprise TEE-DS and provide services for private contracts. A nascent system always lacks user participation and few workers are willing to work, and it requires long time for development. Therefore we choose to build our system on the base of the mature Ethereum system and make use of its availability without any modification to Ethereum.

In addition to Ethereum, we also need to establish a public platform to provide related services such as the client download (both the user client and the worker client) and maintaining the information of TEE-DS. With these services, anyone who has a machine with an enclave can download a worker client, and then find and join TEE-DS as a worker. At the beginning, we may build the TEE-DS with few nodes as a test version. With the broad participation of Ethereum, we anticipate that the requirement of private smart contracts and the remuneration will draw more users and workers in.

Incentive Mechanism. In current design of ShadowEth, workers only get remuneration for executing contracts. But there is no incentive mechanism for storing private contracts. Thus some utilitarian workers may stop their machines immediately after one task, and keep waiting until new tasks are sent to the bounty contract. Since the remuneration is only gained by the fastest worker, the time of execution is closely connected with profit. If a worker only executes contracts without storing them, the latency of synchronizing data (e.g., binary of contracts) from other workers will add more overhead to execution time and reduce expected profit, which motivates workers storing private contracts locally. Furthermore, the worker client is executed as a whole and the response is generated only after all the work is done (including spreading data to other TEE-DS nodes), which is ensured by enclaves, and hence no worker can perform part of the execution to maximize profit.

Workload Measurement. Ethereum provides a gas mechanism, which endows every operation with a fixed price. With Ethereum's runtime environment (EVM), the workload of each execution can be measured and then the gas consumption can be calculated with the gas price. The execution will exit immediately once the gas is run out. This mechanism can prevent DoS attacks such as requests for executing some infinite loop within smart contracts. In our current implementation, the contract is executed in the native environment without a similar monitoring mechanism. To solve this problem, we can measure the workload by execution time. Each invocation task will be endowed with a timeout corresponding to the amount of remuneration. If some workers finish the execution before timeout, the first one will win. If the workload of the execution is so heavy that no worker can finish it before timeout, workers can still gain the remuneration with a measurement to prove that the execution indeed exits after a timeout. The measurement can be generated by the enclave, which is similar to Proof of Elapsed Time (PoET) used by Hyperledger Sawtooth[11].

8 Related Work

Several proposals address the privacy issues of blockchain systems. The cryptocurrencies such as Monero and Zcash improve the confidentiality to some degree. Most of them totally depend on cryptographic methods to hide the information of transactions but with notable limitation such as high computing overhead or partial confidentiality. And they forgo programmability and cannot support smart contracts.

Hawk[5] and Town crier[12] use Intel SGX as a technology to improve the privacy of off-chain contracts. Hawk is a decentralized smart contract framework that processes financial transactions off-chain and

hides the secret on the blockchain, thus providing transaction privacy. It divides the smart contract into two parts: the private portion and the public portion, and the execution is facilitated by a special party called manager which is protected by SGX. In contrast to Hawk, numerous workers in ShadowEth comprise a network for execution and storage of private contracts, which provides increased reliablity. Towncrier is a data feed system that serves as a high-trust bridge between Ethereum blockchain and existing websites. It retrieves website data, serves it to contracts in need on the blockchain, and enables private data requests with encrypted parameters. It executes its core functionality inside an SGX enclave to protect data against malicious attackers.

Coco is a high-scale, confidential blockchain framework. It is a license chain other than public chain and is designed for enterprise requirements. Each node in the Coco network is protected by TEE such as Intel SGX and verified before joining in the network. Therefore each node is assumed to be not malicious and there is no need to defend against Byzantine faults. The advantage is that Coco can adopt simple and quick consensus algorithm such as Raft instead of wasteful and compute-intensive algorithms like PoW. Therefore Coco can provide higher throughput and lower latency than general public blockchains such as Ethereum. Also, TEE can guarantee the confidentiality of transactions and smart contracts. However, Coco is not suitable for building public blockchain systems in that not everyone is free to join. In contrast, ShadowEth is built on existing public blockchains like Ethereum and can provide higher fault-tolerance.

TEEchan[13] and TEEchain[14] establish high-performance and secure micropayment channel for the Bitcoin network. Users that need frequent mutual transactions can set up a channel and perform fund transfer through the channel without sending transactions onto the blockchain, which can improve the throughput and lower the latency. Only the setup and settlement will be reflected as transactions on the chain. TEEchan and TEEchain leverage TEEs to guarantee the security and the confidentiality of the channel without any modification to the Bitcoin network. But they only provide enhancement for simple bitcoin transactions while ShadowEth applies to any smart contract system.

9 Conclusions

We introduced ShadowEth, a system that addresses a major concern about the current smart contract systems based on the blockchain — the lack of confidentiality. Unlike the pure and computationally complex solution (such as Zcash) and the reconstructed licensing chain (such as Coco), ShadowEth can guarantee the confidentiality of existing public blockchains like Ethereum without any modification. ShadowEth separates the process of the verification of a smart contract from the private execution, and only puts the verification onto the blockchain, without revealing any secrets. The actual logic of smart contracts is executed by off-chain TEE and the communication is encrypted by a secret key only kept by TEE. We used a native Ethereum smart contract named bounty contract to handle the publishing, verification and settlement of a private contract, which ensures the integrity and coerciveness. Then we presented the applicability by case studies. We also implemented a prototype using Intel SGX on the Ethereum network and analyzed the security and availability. We believe that ShadowEth is a practical approach to building a confidential public smart contract system.

References

[1] Meiklejohn S, Pomarole M, Jordan G, Levchenko K, McCoy D, Voelker G, Savage S. A fistful of bitcoins: Characterizing payments among men with no names. In *Proc. the Conf. Internet Measurement Conf.*, October 2013, pp.127-140.

[2] Ron D, Shamir A. Quantitative analysis of the full bitcoin transaction graph. In *Proc. the 17th International Conf. Financial Cryptography and Data Security*, April 2013, pp.6-24.

[3] Parno B, Howell J, Gentry C, Raykova M. Pinocchio: Nearly practical verifiable computation. In *Proc. IEEE Symp. Security and Privacy*, May 2013, pp.127-140.

[4] Miers I, Garman C, Green M, Rubin A D. Zerocoin: Anonymous distributed E-cash from bitcoin. In *Proc. IEEE Symp. Security and Privacy*, May 2013, pp.397-411.

[5] Kosba A, Miller A, Shi E, Wen Z K, Papamanthou C. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Proc. IEEE Symp. Security and Privacy*, May 2016, pp.839-858.

[6] Costan V, Devadas S. Intel SGX explained. IACR Cryptology ePrint Archive: Report 2016/086, 2016. http://eprint.iacr.org/, Mar. 2018.

[7] Xu Y Z, Cui W D, Peinado M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proc. IEEE Symp. Security and Privacy*, May 2015, pp.640-656.

[8] Shih M W, Lee S, Kim T, Peinado M. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proc. the Annual Network and Distributed System Security Symposium*, March 2017.

[9] Shinde S, Chua Z L, Narayanan V, Saxena P. Preventing page faults from telling your secrets: Defenses against pigeonhole attacks. In *Proc. the 11th ACM on Asia Conf. Computer and Communications Security*, May 2016, pp.317-328.

[10] Lee S, Shih M W, Gera P, Kim T, Kim H, Peinado M. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proc. the 26th USENIX Security Symp.*, August 2017, pp.16-18.

[11] Prisco G. Intel develops 'Sawtooth Lake' distributed ledger technology for the Hyperledger project. https://bitcoinmagazine.com/articles/intel-develops-sawtooth-lake-distributed-ledger-technology-for-the-hyperledger-project-14603974-61/, Mar. 2018.

[12] Zhang F, Cecchetti E, Croman K, Juels A, Shi E. Town crier: An authenticated data feed for smart contracts. In *Proc. the 23rd ACM SIGSAC Conf. Computer and Communications Security*, October 2016, pp.270-282.

[13] Lind J, Eyal I, Pietzuch P, Sirer G S, Shi E. Teechan: Payment channels using trusted execution environments. arXiv preprint arXiv: 1612.07766, 2016. http://arxiv.org/abs/1612.07766, Mar. 2018.

[14] Lind J, Eyal I, Kelbert F, Naor O, Pietzuch P, Sirer G S. Teechain: Scalable blockchain payments using trusted execution environments. arXiv preprint arXiv: 1707.05454, 2017. http://arxiv.org/abs/1707.05454, Mar. 2018.

Rui Yuan is currently a postgraduate student of the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai. His research interests include blockchain and system security.

Yu-Bin Xia received his B.S. degree in computer science from Fudan University, Shanghai, in 2004, and Ph.D. degree in computer science from Peking University, Beijing, in 2010. He is currently an associate professor of the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai. He is a senior member of CCF, and a member of ACM and IEEE. His research interests include system software, computer architecture, and system security.

Hai-Bo Chen received his B.S. and Ph.D. degrees in computer science from Fudan University, Shanghai, in 2004 and 2009, respectively. He is currently a professor of the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai. He is a distinguished member of CCF, and a senior member of ACM and IEEE. His research interests include software evolution, system software, and computer architecture.

Bin-Yu Zang received his Ph.D. degree in computer science from Fudan University, Shanghai, in 1999. He is currently a professor and the director of the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai. He is a distinguished member of CCF, and a member of ACM and IEEE. His research interests include compilers, computer architecture, and systems software.

Jan Xie is a co-founder and CEO of Cryptape Inc., a blockchain company in Hangzhou.