2018 IEEE 25th International Conference on High Performance Computing (HiPC)

Decentralized Privacy-preserving Timed Execution in Blockchain-based Smart Contract Platforms

Chao Li
School of Computing and Information
University of Pittsburgh
Pittsburgh, USA
Email: chl205@pitt.edu

Balaji Palanisamy
School of Computing and Information
University of Pittsburgh
Pittsburgh, USA
Email: bpalan@pitt.edu

***Abstract*—Timed transaction execution is critical for various decentralized privacy-preserving applications powered by blockchain-based smart contract platforms. Such privacy-preserving smart contract applications need to be able to securely maintain users' sensitive inputs off the blockchain until a prescribed execution time and then automatically make the inputs available to enable on-chain execution of the target function at the execution time, even if the user goes offline. While straight-forward centralized approaches provide a basic solution to the problem, unfortunately they are limited to a single point of trust. This paper presents a new decentralized privacy-preserving transaction scheduling approach that allows users of Ethereum-based decentralized applications to schedule transactions without revealing sensitive inputs before an execution time window selected by the users. The proposed approach involves no centralized party and allows users to go offline at their discretion after scheduling a transaction. The sensitive inputs are privately maintained by a set of trustees randomly selected from the network enabling the inputs to be revealed only at the execution time. The proposed protocol employs secret key sharing and layered encryption techniques and economic deterrence models to securely protect the sensitive information against possible attacks including some trustees destroying the sensitive information or secretly releasing the sensitive information prior to the execution time. We demonstrate the attack-resilience of the proposed approach through rigorous analysis. Our implementation and experimental evaluation on the Ethereum official test network demonstrates that the proposed approach is effective and has a low gas cost and time overhead associated with it.**

I. Introduction

In the age of big data, blockchain [32] has become a promising technology to enable decentralized protection of data integrity [18] and ensuring data quality [7]. Any data stored in a blockchain is backed up and verified by all the nodes in the network and provides a strong resilience against attacks that can tamper the integrity of the data. With this great feature offered by blockchains, recent implementations of blockchain-based smart contract platforms, such as Ethereum [46] and NEO [33], have attracted a large number of developers to build decentralized applications using smart contracts that avoid the need of a centralized server to manage and maintain the data [6], [30], [31]. The market cap for the leading smart contract platform, Ethereum, peaked at $134 billion [15] in 2018 and thousands of decentralized applications, ranging from social networks to financial software, have been developed over Ethereum [41]. The Smart Contracts market is estimated to grow at a CAGR of 32% during the period 2017 to 2023 [40].

A decentralized application may involve one or more smart contracts and each smart contract may contain multiple functions that need to be invoked by application users through transactions. For instance, a sealed-bid auction smart contract [47] requires bidders to reveal their sealed bids by invoking a function (e.g., a *reveal() function*) during a time window. Similarly, a voting smart contract [30] requires voters to publish their votes using a *vote()* function during the voting time window. Each called function in a smart contract is executed by the entire blockchain network. Since both function code and function inputs (i.e., bid or vote) are available on the blockchain, the function outputs are deterministic and their correctness can be verified by the network, thus cutting out centralized middlemen or intermediaries for running these functions [24].

A key fundamental limitation of existing smart contract platforms is the lack of support for users to schedule timed execution of transactions such that their target functions can be invoked at a later time, even when the users go offline. For example, if Bob plans to take a week off work and could not respond to an auction or voting mechanism implemented on Ethereum during the prescribed time windows, he needs a mechanism to schedule these timed transactions by automatically invoking *reveal()* and *vote()* during the time windows. Here, the inputs to these functions namely the bids and the votes are extremely sensitive and need to be securely protected until the prescribed time windows even when Bob is offline. Scheduling timed execution of functions is common in centralized application environments. For instance, Boomerang [11] allows users of Gmail to schedule their emails to be sent when users have no connection with the Internet. Similarly, Postfity [35] helps users to schedule messages to be posted onto many centralized social networks. However, a centralized approach to supporting timed execution of transactions incurs a single point of trust and violates the key design principle of decentralization inherent in blockchain-based smart contract platforms [43]. In general, the design of timed execution of transactions in decentralized platforms such as Ethereum is

978-1-5386-8386-6/18/$31.00 ©2018 IEEE
DOI 10.1109/HiPC.2018.00037

challenged in two aspects. First, when a transaction invoking a function is deployed into the network, the invoked function is executed immediately which makes it difficult to support timed execution when the user has already gone offline. Another key challenge arises due to privacy concerns associated with the input data to the function. To guarantee verifiability of function outputs, function inputs need be put onto the blockchain and as a result, both function inputs and outputs become public to all peers at the time the schedule is initialized leading to privacy risks with the input data. The proposed privacy-preserving timed-execution approaches find numerous applications in high performance computing. For instance, recent projects such as Golem [2] and iEx.ec [3] focus on developing decentralized supercomputers and high performance computing platforms without vendor lock-in. These solutions leverage the Ethereum as a marketplace application to link buyers and sellers of computing resources without requiring an intermediary. When smart contracts are used to manage and schedule computing tasks in such platforms, privacy-preserving timed-execution techniques can effectively protect privacy of sensitive inputs of the scheduled and in-queue computing tasks.

In this paper, we design and develop a new decentralized privacy-preserving timed execution mechanism that allows users of Ethereum-based decentralized platforms to schedule timed execution of transactions without revealing function inputs and outputs prior to the execution time selected by the users. The proposed approach is decentralized and involves no centralized party and does not include any single point of trust. After transactions have been scheduled, it requires no further interaction from users and allows users to go offline at their discretion. The mechanism does not reveal function inputs before the execution time window selected by a user as function inputs are privately maintained by a set of trustees randomly selected from the network and released only during the execution time window. The function inputs are protected through secret share [39] and multi-layer encryption [12] and possible misbehaviors of the trustees are made detectable and verifiable through a suit of misbehavior report mechanisms implemented in the Ethereum Smart Contracts and any verified misbehavior incurs monetary penalty on the violator. We implement the proposed approach using the contract-oriented programming language *Solidity* [42] and test it on the Ethereum official test network *rinkeby* [36] with Ethereum official Go implementation *Geth* [20]. Our implementation and experimental evaluation that the proposed approach is effective and the protocol has a low gas cost and time overhead associated with it.

II. Overview of timed execution in Ethereum

In this section, we first present the preliminaries of the Ethereum smart contract platform [46] and describe the challenges involved in implementing timed execution of smart contracts over Ethereum. We then present the key ideas behind the proposed solution and introduce the organization of the proposed protocol and discuss the security challenges and potential attacks encountered in the proposed approach.

A. Preliminaries

A blockchain represents a decentralized and distributed public digital ledger that guarantees that the records stored in it cannot be tampered without compromising a majority of nodes in the network [32]. Then, a smart contract is a piece of program code stored in a blockchain and it usually consists of multiple functions. In the leading smart contract platform Ethereum [46], there are two types of accounts, namely External Owned Accounts (EOAs) controlled by private keys and Contract Accounts (CAs) for storing smart contract code. An Ethereum node can create as many as EOAs and then use EOAs to create CAs by deploying smart contracts. However, since smart contracts are passive, their execution must be invoked through either a transaction sent by an EOA or a message sent from a CA. As a result, the transactions/messages, as well as function inputs inside them, are all recorded by the Ethereum blockchain, which makes the function outputs deterministic because all Ethereum nodes can execute the function with the same inputs and gets the same outputs. In Ethereum, to deploy a smart contract (i.e., CA) or call a smart contract function changing any data on blockchain, one needs to pay for Gas [46]. Gas can be exchanged with Ether, the cryptocurrency used in Ethereum, and Ether can be exchanged with real money.

B. Problem statement

The Ethereum blockchain platform [46] can be viewed as a giant global computer as shown in Figure 1. If a user creates a EOA and uses the EOA to send a transaction with inputs x_1 and x_2 to call function $f(x_1, x_2)$ of a smart contract C at time t_1, function $f(x_1, x_2)$ will be executed instantly and the inputs x_1 and x_2 will be made public. This is acceptable if the user just wants to reveal x_1 and x_2 at time t_1. However, if the user needs to reveal x_1 and x_2 during a future execution time window w_e, sending the transaction at t_1 will not work. For example, Bob may want to make function $reveal(amount, nonce)$ of a sealed-bid auction smart contract [47] be executed during a future execution time window w_e. Then, sending the transaction out at t_1 will make his bid value be known to all other bidders immediately, which violates his privacy requirements.

C. Privacy-preserving timed execution

To support privacy-preserving timed execution of smart contracts, the transaction calling function $f(x_1, x_2)$ must be sent during the prescribed execution time window w_e while inputs x_1 and x_2 should not be revealed before w_e. Our proposed protocol for supporting privacy-preserving timed execution is implemented as two smart contracts, namely a unique scheduler contract C_s managing all schedule requests of users in Ethereum and a proxy contract C_p deployed by each user having a schedule request. At the time of setting a timed execution, the protocol requires the user to (1) store

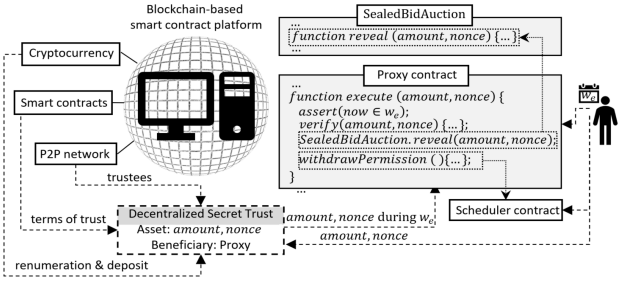


Fig. 1: At time t_1, Bob wants to schedule function *reveal(amount,nonce)* in contract *SealedBidAuction* [47] to be executed during a future time window w_e

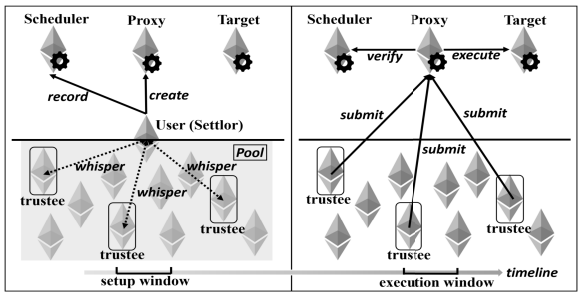


Fig. 2: Protocol overview

schedule information, including a cryptographic *keccak-256* hash [8] of function inputs x_1 and x_2 to the scheduler contract C_s, (2) deploy a proxy contract C_p and (3) employ a group of EOAs as trustees. The main functionality of the proxy contract C_p is implemented through a function $execute()$ in it. Once C_p receives a transaction during w_e with the desired inputs x_1 and x_2 verified through their hashes in scheduler contract C_s, the function $execute()$ will immediately send a message calling the target function $f(x_1, x_2)$ with inputs x_1 and x_2. The trustees are in charge of storing inputs x_1 and x_2 off the blockchain before the execution time window w_e and they send a transaction with the inputs to the proxy contract C_p during w_e. The terms of the decentralized secret trust created by the user as a settlor, namely what the trustees can or cannot do, are programmed as functions in smart contracts C_s and C_p. Each trustee needs to pay a security deposit d (i.e., Ether) to the scheduler contract C_s and any detectable misbehavior of this trustee makes the deposit be confiscated. The security deposit serves as an economic deterrence model for enforcing behaviors of peers in the blockchain network [5], [31]. Finally, after the trustees have sent a transaction with inputs x_1 and x_2 to the proxy contract C_p during w_e, they can withdraw both their deposit and remuneration paid by the user from the scheduler contract C_s. In the example of Figure 1, at t_1, Bob stores hash of inputs $amount$ and $nonce$ to C_s, deploys C_p and employs a group of trustees. These trustees, after signing an agreement with Bob, are in charge of revealing the asset $amount$ and $nonce$ to the beneficiary, proxy contract C_p, during w_e. During the execution time window w_e, after the trustees have sent a transaction with inputs $amount$ and $nonce$ to C_p, the function $execute()$ in C_p can trigger $reveal()$ in the *SealedBidAuction* contract through $SealedBidAuction.reveal()$ and also unlock trustees' deposit and remuneration in C_s through $withdrawPermission()$.

D. Protocol overview

The proposed protocol consists of four components:

Trustee application: At any point in time, an EOA can apply to C_s for getting added into a trustee candidate pool maintained by C_s by submitting its working time window and paying a security deposit. During the working time window, the EOA should be able to connect with Ethereum to send transactions to the proxy contract C_p. In the example shown in Figure 2, we notice that ten EOAs joined the pool. The public pool then makes the entire network learn that this EOA can provide services during its declared working times.

User schedule: During setup time window w_s, a user can schedule a transaction by registering the schedule to scheduler contract C_s, deploying a proxy contract C_p, and secretly selecting trustees from the pool. The selected trustees should keep the function inputs privately before the execution time window w_e while revealing them during w_e to make the target function be executed. In Figure 2, during setup window w_s, the user informed the schedule with the scheduler contract C_s and deployed the proxy contract C_p. Then, the user randomly selected three EOAs from the pool as trustees and signed agreements with the trustees through private channels created by the whisper protocol [45]. Any data exchanged through the whisper channels are encrypted and can only be viewed by the data sender and data recipient.

Function Execution: During execution time window w_e, the selected trustees submit the function inputs to the proxy contract C_p through transactions, which triggers C_p to verify correctness of function inputs with C_s and then call the scheduled function in the target contract C_t. In Figure 2, during w_e, the trustees submitted stored data to proxy contract C_p. After verifying the received data with the hashes stored in scheduler contract C_s, C_p called the function in C_t.

Misbehavior report: During the entire process, trustees may perform several types of misbehaviors violating the protocol, such as secretly disclosing stored data before w_e or rejecting to submit stored data during w_e. To tackle these issues, the protocol involves several misbehavior report mechanisms that allow any witness of a misbehavior to report it to the scheduler contract C_s and earn a component of the deposit paid by the suspect trustee once the report is verified to be true.

E. Security challenges and attack models

The proposed mechanism encounters several critical security challenges, which can be roughly classified using two attack models.

Time difference attacks: The time difference attack happens when an adversary aims at obtaining the function inputs at a time point t_d earlier than the execution time window w_e so that he can leverage the time difference between t_d and w_e to achieve his purpose. There are two key methods to launch a time difference attack.

- **Trustee identity disclosure**: In *user schedule* component of the protocol, trustees are secretly selected by user U. Therefore, from the perspective of EOAs besides the selected trustees and user U, all EOAs in network with working time windows satisfying U's requirement have equal chance to be selected by U, thus protecting the identifications of selected trustees with highest entropy and uncertainty. However, a trustee, after being selected, may announce its identity to the public to seek trade with potential adversaries about the stored data. To prevent such misbehavior, the proposed protocol employs a trustee identity disclosure report mechanism in *misbehavior report* component of the protocol, which forces a trustee to disclose its identity with the sacrifice of the confiscation of its security deposit.
- **Advance disclosure**: A trustee may choose to voluntarily disclose the stored data to the entire network without seeking bribery. To penalize such misbehavior, an advance disclosure report mechanism is employed in the *misbehavior report* component, which makes any trustee disclosing its stored data in advance lose its security deposit.

Execution failure attack: The execution failure attack happens when an adversary aims at making the execution of the target function fail during the execution time window w_e. There are two key methods to launch this attack.

- **Absent trustee**: A trustee may become absent during the execution time window w_e, which makes its stored data get lost. To prevent this type of misbehavior, the *user schedule* component of protocol requires each selected trustee to provide a signature, which will only be revealed along with the function inputs during w_e. Therefore, before w_e, the identities of trustees are kept secret. In contrast, during w_e, the identities become public so that any present trustee can report an absent trustee through the absent trustee report mechanism in the *misbehavior report* component of protocol, which penalizes any absent trustee by confiscating its security deposit.
- **Fake submission**: A trustee may submit fake stored data to the proxy contract C_p during w_e, which may cause the restoration of the function inputs to fail. The protocol handles this type of misbehavior using the fake submission report mechanism in the *misbehavior report* component of protocol, which confiscates violator's security deposit if its submission is proved to be fake.

III. Protocol description

In this section, we present the proposed protocol organized along the four components introduced in Section II-D.

A. Trustee application

The first component *trustee application* allows EOAs that want to earn remuneration through the trustee job to register to the scheduler contract C_s and make their information public. There are three key steps in this component. We note that a step with a gray bullet (e.g., 1) refers to an off-chain action not recorded by blockchain while a step with a white bullet (e.g., 2) refers to an on-chain action recorded by blockchain. We will distinguish off-chain and on-chain steps with the two bullet types in all four components of the protocol.

Trustee application
Input: scheduler contract C_s
Apply:
1 An Ethereum node creates a new EOA.
2. This EOA applies to the scheduler contract C_s for being added into the trustee candidate pool by submitting a public key, a whisper key, working time window, a security deposit and a beneficiary address.
3. The scheduler contract C_s verifies the application and accept the application if all required data has been submitted.

Step 1: Each trustee candidate should be a newly generated EOA, which only has an amount of Ether (the cryptocurrency in Ethereum) that will be submitted to the scheduler contract C_s as security deposit d in step 2. No additional Ether should be left because we will need the account to make its account private key public during execution time window w_e.
Step 2-3: An EOA should apply for the trustee candidate by sending a transaction to C_s with the five listed information.

- The public key will later be used by user U in step 8 of *user schedule* component to generate $onions$ [12]. Here, the term $onion$ refers to the output of iteratively encrypting data with multiple public keys.
- The whisper key will later be used by user U in *user schedule* component to establish private channel with this EOA through whisper protocol [45].
- The working time window will be used by user U in step 6 and 10 of *user schedule* component to select trustees satisfying U's requirements (i.e., execution time window).
- The security deposit is a fixed amount of Ether hard-coded in scheduler contract C_s. Once being submitted to C_s, the deposit can only be withdrawn at the end of EOA's working time window, if there is no misbehavior reported through report mechanisms in *misbehavior report* components.
- Finally, the protocol needs the EOA to make its account private key public in *function execution* component, so the beneficiary address will be the address of a safe EOA to receive deposit and remuneration withdraw.

B. User schedule

The second component *user schedule* prescribes how a user should set a schedule through three key operations, namely deploying a proxy contract (step 3), registering the schedule information to scheduler contract C_s (step 4) and implementing a two-round trustee selection (step 5-13).

For the illustration of the protocol in step 5 to 13, we will use the example shown in Figure 3.

User schedule

Input: scheduler contract C_s, target contract C_t

Initialization:

1 User U decides function inputs IN, execution time window w_e, secret sharing parameters (m, n), number of layers l, a 256-bit secret key key and a 256-bit random number R_U.
2 User U computes the remuneration r.
3. User U deploys proxy contract C_p to the network.
4. User U registers the schedule to scheduler contract C_s with $(w_e, m, n, l, C_p^{addr}, r)$ and receive a schedule ID sid.
5 User U splits key to n $shares$ through (m, n) secret sharing.

First-round trustee selection:

6 User U randomly selects $n(l-1)$ trustees and sends each trustee a (sid, tid), where tid refers to a non-repeated ID in the range of $[0, n(l-1))$ assigned to the trustee.
7 Each selected trustee T then does the following:
  7.1. Verify $(U^{addr}, sid, tid, w_e, r)$ with C_s.
  7.2. Generate a 256-bit random number R_T.
  7.3. Take keccak256 hash $h(T^{addr}, R_T)$.
  7.4. Sign $(U^{addr}, sid, tid, h(T^{addr}, R_T))$ with T's private key, which gives signature $vrs = (v, r, s)$.
  7.5. Send $h(T^{addr}, R_T)$ and vrs back to U.
8 User U encrypts $shares$ to $onions$ with public keys of selected trustees.
9. User U takes keccak256 hash $h(onion)$ of each $onion$ and submits the hash values to C_s.

Second-round trustee selection:

10 User U randomly selects n trustees and sends each trustee a $(sid, tid, onion)$, where tid is non-repeated in $[n(l-1), nl)$.
11 Each selected trustee T follows step 7, but in addition verifying received $onion$ with $h(onion)$ in C_s.

Ciphertext and hash disclosure:

12 User U encrypts (IN, vrs, R_U) with key and make $E(key, (IN, vrs, R_U))$ public.
13. User U submits keccak256 hash $h(IN, R_U)$ and each trustee's $h(T^{addr}, R_T)$ to C_s.

Step 2: The total remuneration that should be paid by user U is $r = nlr_t + r_e$, where r_d is a fixed per trustee remuneration hard-coded in C_s and r_e is a fixed amount of reward hard-coded in C_m paying to the first trustee calling $execute()$ in C_t during w_e. Both r_d and r_e can only be withdrawn by trustees after the end of execution time window w_e.

Step 4: After the schedule has been registered in C_s, the on-chain schedule information cannot be modified. Therefore, the information can be used by trustee candidates later in step 7 and 11 to verify the information transmitted through off-chain whisper channels from user U.

Step 5: The Shamir secret sharing scheme [39] with parameter (m, n) can split the key to n $shares$. Later, any m $shares$ among the n can be combined to restore the key while even $m-1$ $shares$ fail to do it. Therefore, even if some $shares$ are compromised, the compromised $shares$ may be insufficient to restore the key before execution window w_e while the rest $shares$ may still be sufficient to restore the key during w_e. In the example of Figure 2, we set $(m, n) = (2, 3)$, so three $shares$ are generated from key after splitting.

Step 6-13: The design of two-round trustee selection implements the decentralized secret trust. The trustees selected in the first round should agree the user encrypt the $shares$ with their public keys for multiple layers so that the $shares$ become $onions$ [12] and harder to be compromised. Then, the trustees selected in the second round should take charge of storing these $onions$. Later, during w_e, once both the private keys of the first-round trustees and $onions$ stored by the second-round trustees are made public, the key can be restored to decrypt the function inputs. The process offers following additional security features:

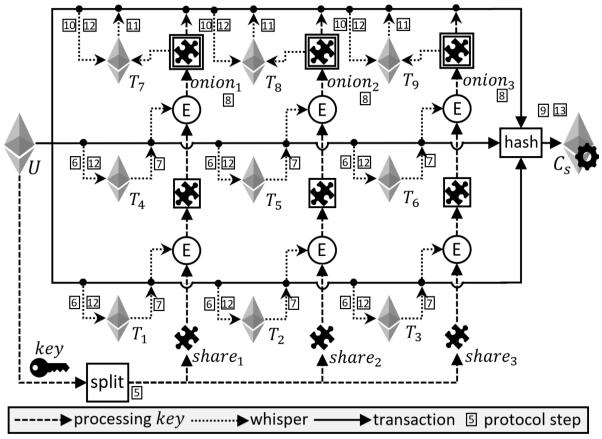


Fig. 3: User schedule example

- The identities of selected trustees are kept private. In these steps, each trustee only communicates with the user through a whisper channel and all information that needs to be made public are announced by the user (step 9,12,13). Therefore, the identity of each trustee is only known to the user. This feature helps in suppressing collusion among trustees.
- The identities of selected trustees are verifiable and only the trustees can pass the verification. To be verified as a specific trustee, both the trustee's address T^{addr} and the nonce R_T need to be submitted to C_s and their hash should match with the one submitted by user in step 13. Since R_T is created by the trustee, only the trustee has the ability to pass the verification. This feature also helps in suppressing collusion among trustees. We will discuss it in detail later in *misbehavior report* component.
- The identities of selected trustees are undeniable. The user has signatures of the trustees (step 7,11) and the encrypted signatures are made public in step 12. Therefore, once key is restored during w_e, the decrypted signatures can reveal the identities of all trustees. This feature helps in detecting absent trustees who disappear during w_e.
- The trustees are also protected against adversaries. It may be insecure to only allow users to publicly speak. Such a user may fabricate information and make trustees lose security deposit. To protect trustees from such users. Once a user has registered a schedule in step 4, the submitted information cannot be changed. Then, in step 7 and 11, each trustee can check the information before sending a signature to the user. This is also the main reason that we need two rounds.

In step 11, the second-round trustees should first verify the correctness of the onions with the hash submitted by the user in step 9 and then provide signatures.

In the example of Figure 3, six trustees (T_1-T_6) are selected by user U in the first round and their six public keys encrypt each of the three $shares$ with two layers, thus turning the $shares$ into two-layer $onions$. Then, three trustees (T_7-T_9) are selected by user U in the second round to store the three $onions$. Finally, U ends the schedule by making the ciphertext public and submitting all hash values to C_s.

C. Function Execution

The third component of the protocol, *function execution* indicates how the trustees selected in *user schedule* component should collaboratively reveal the function inputs during execution window w_e and send a transaction with the function inputs to the proxy contract C_p through two phases, namely *submission* (step 1-2) and *execution* (step 3-6).

Function Execution

Input: scheduler contract C_s

Submission (first half of w_e):

1. Each trustee T verifies its identity with $h(T^{addr}, R_T)$ by submitting R_T to C_s.
2. Each trustee T submits $onion$ or its private key to C_s, where $onion$ should be verified with $h(onion)$.

Execution (second half of w_e):

3. Any trustee T can get $shares$ by decrypting $onions$ with the private keys.
4. Any trustee T can get key by combing any m $shares$.
5. Any trustee T can get (IN, vrs, R_U) by doing $D(key, E(key, (IN, vrs, R_U)))$.
6. Any trustee T can submit (IN, R_U) to proxy contract C_p, where (IN, R_U) can be verified with $h(IN, R_U)$ in C_s and the correct function inputs IN will trigger C_p to call the target contract C_t.

Step 1-2: The *submission* phase indicates the first half of execution window w_e, during which the protocol requires first-round and second-round trustees to submit their private keys and stored $onions$, respectively. To submit either a private key or an $onion$, a trustee should also provide the nonce R_T generated in step 7 and 11 of *user schedule* so that its identity can be verified with $h(T^{addr}, R_T)$.

Step 3-6: The *execution* phase refers to the second half of execution window w_e. Since both $onions$ and private keys have been submitted, during this phase, any verified trustee should be able to turn $onions$ back to $shares$. Then, based on Shamir secret sharing scheme, any m shares can be combined to restore the key created by user S in step 1 of *user schedule*. After getting the key, any trustee is able to decrypt the encrypted (IN, vrs, R_U). Finally, before the end of w_e, a verified trustee, after obtaining function inputs IN and nonce R_U, should send proxy contract C_p a transaction with both IN and R_U. Then, C_p will immediately verify received IN and R_U with $h(IN, R_U)$ in scheduler contract C_s. If both of them are correct, C_p immediately send a message with IN to the target contract C_t to call the scheduled function.

D. Misbehavior report

The *misbehavior report* represents the final component of the protocol and involves four types of misbehaviors that will result in the violator's security deposit being confiscated. All these misbehaviors are witnessable and the protocol rewards the reporter of a misbehavior a component of the violator's security deposit as an incentive while sending the rest of the violator's security deposit to the user.

Misbehavior report

Input: scheduler contract C_s

Trustee identity disclosure report:

1. Before the start of execution time w_e, any EOA can report a trustee identity disclosure misbehavior by submitting the nonce R_T of the violator to scheduler contract C_s.
2. If $h(T^{addr}, R_T)$ using the submitted R_T is same as the one in C_s, the misbehavior is verified.

Advance disclosure report:

3. Before the start of execution time w_e, any EOA can report an advance disclosure misbehavior by submitting the private key belonging to the violator to scheduler contract C_s.
4. If the public key derived from that private key is same as the violator's public key in C_s, the misbehavior is verified.

Absent trustee report:

5. After step 5 in *function execution*, any trustee can report an absent trustee misbehavior to scheduler contract C_s by submitting the signature vrs of the absent trustee.
6. The address of the violator can be derived through $T = sigVerify((U^{addr}, sid, tid, h(T^{addr}, R_T)), vrs)$.

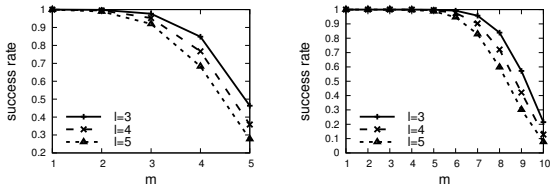
Fake submission report:

7. After step 2 in *function execution*, any trustee can report a fake submission misbehavior to scheduler contract C_s if the trustee finds a submitted private key is incorrect.
8. If the public key derived from that private key is different from violator's public key in C_s, the misbehavior is verified.

Trustee identity disclosure report: This report mechanism is designed to handle the trustee identity disclosure misbehavior presented in Section II-E. Before the start of execution window w_e, a trustee may choose to reveal its identity to seek collusion. To prove its identity, the violator has to reveal the nonce R_T created by itself in step 7/11 of *user schedule* so that its identity can become verifiable through $h(T^{addr}, R_T)$ in C_s. However, with this report mechanism, any EOA, after knowing R_T before w_e, can report it to C_s to earn reward.

Advance disclosure report: The advance disclosure misbehavior introduced in Section II-E can be handled using this report mechanism. Before the start of w_e, a round-one trustee may choose to disclose its private key, which may help an adversary to decrypt $onions$ to $shares$, restore key and obtain IN before the start of w_e. However, with this report mechanism, any EOA, after knowing violator's private key before w_e, can betray the violator by reporting it to C_s.

Absent trustee report: This report mechanism handles the absent trustee misbehavior described in Section II-E. Any trustee may become absent during w_e, thus increasing the failure chance of schedule. With this report mechanism, any trustee, after obtaining signatures of all other trustees in step



(a) $n = 5$ (b) $n = 10$

Fig. 4: Schedule success rate when 5% of trustees perform misbehaviors inadvertently

Component	Step	Function	Purpose
Schedule	5	share	split key to $shares$
	7,11	ecsign	sign data with private key
	8	encrypt	encrypt $shares$ to $onions$
	9,13	solidity$Sha3$	compute keccak256 hash
Execute	3	combine	combine $shares$ to key
	4	decrypt	decrypt $onions$ to $shares$

TABLE I: Key off-chain functions in node.js, *share()* and *combine()* are in secrets.js [38], *ecsign()* is in ethereumjs-util [16], *encrypt()* and *decrypt()* are in eth-ecies [14], *solidity$Sha3$()* is in web3-utils [44]

5 of *function execution*, can locally verify attendance of all other trustees, thus being able to report absent trustees to C_s.

Fake submission report: Finally, the design of fake submission report aims at dealing with the fake submission misbehavior presented in Section II-E. In step 2 of *function execution*, a submitted private key may not be the right one. Any trustee can locally verify a private key submitted by a suspect trustee through deriving the corresponding public key from the private key and comparing it with the public key submitted by that suspect trustee during *trustee application*, thus becoming able to report violators to C_s.

IV. Security analysis

Next, we analyze the security guarantees of the proposed approach based on the rational adversary model. Recently, it has been widely recognized that assuming an adversary to be semi-honest or malicious is either too weak or too strong and hence modeling adversaries with rationality is a relevant choice in several attack scenarios [13], [34]. Informally, a semi-honest adversary follows the prescribed protocol but tries to glean more information from available intermediate results while a malicious adversary can take any action for launching attacks [21]. A rational adversary lies in the middle of the two types. That is, rational adversaries are self-interest-driven, they choose to violate protocols, such as colluding with other parties, only when doing so brings them a higher profit. In this paper, in order to design our approach with strong and practical security guarantees, we model all EOAs to be rational adversaries without assuming any of them to be honest or semi-honest.

Without countermeasures, such rational adversaries, after being selected by user as trustees, may perform four types of misbehaviors introduced in Section II-D, including *trustee identity disclosure*, *advance disclosure*, *absent trustee* and *fake submission*. As per the four misbehavior report mechanisms designed in *misbehavior report* component, as long as the key can be restored during the execution time window w_e, any of the four types of misbehaviors will lead to confiscation of the violator's deposit. To prevent restoration of the key so that misbehaviors can be performed in free, a certain fraction of trustees must collude to not submit their stored data (i.e., $onion$ or private key) together. However, due to trustee identity disclosure report mechanism in *misbehavior report*, revealing trustee identity to other EOAs means losing deposit, so such a collusion will not happen among rational adversaries.

It is possible that a rational adversary performs misbehaviors inadvertently, such as forgetting providing the service or losing EOA's private key. Such kinds of inadvertent misbehaviors lead to same results of intentionally performing *absent trustee* misbehavior. If we denote the percentage of EOAs performing inadvertent misbehaviors as p_{IM}, the success rate of a schedule with parameters (l, m, n) will be computed through the Cumulative Distribution Function of Binomial distribution, namely $SR = 1 - \sum_{i=n-m+1}^{n} \binom{n}{i} P^i (1-P)^{n-i}$, where $P = 1-(1-p_{IM})^l$ represents the probability that one $share$ is lost. In Figure 4, we present the computed schedule success rate when 5% of trustees perform misbehaviors inadvertently. Specifically, in Figure 4(a), by fixing n to 5 and changing m from 1 to 5, it shows that a smaller m, namely lower threshold for restoring key, performs higher resistance against inadvertent misbehaviors. By further changing l from 3 to 5, we can find that a smaller l offers better resistance against inadvertent misbehaviors. Then, in Figure 4(b), n is increased to 10. The increment of n enhances the resistance against inadvertent misbehaviors when m and l do not change. Thus, larger l and n while smaller m help maintaining high resistance against inadvertent misbehaviors.

V. Implementation

In this section, we present the implementation of the proposed protocol and discuss the experimental evaluation of the proposed mechanism in Ethereum.

A. Implementation of protocol

We first introduce the implementation setup and then present both key off-chain functions in node.js and on-chain functions in *Solidity* [42] and demonstrate how they work in practice. After that, we present two test instances used in our experimental evaluation.

Setup: We programmed the smart contracts in *Solidity* [42], the most commonly used smart contract programming language, deployed them to the Ethereum official test network *rinkeby* [36] and tested them with Ethereum official Go implementation *Geth* [20]. Our experiments are performed on an Intel Core i7 2.70GHz PC with 16GB RAM.

Implemented functions: The protocol primarily relies on 6 off-chain functions shown in Table I and 15 on-chain functions

Component	Step	Function	Purpose
Apply	2,3	newCandidate	join candidate pool
Schedule	4	newUser	register as a new user
	4	newSchedule	initialize a new schedule
	9	setOnion	submit hashes of onions
	13	setTrustee	submit hashes of trustees
Execute	1,2	submitPrivkey	submit private key
	1,2	submitOnion	submit onion
	6	execute	execute the target contract
	7	withdrawD	withdraw security deposit
	7	withdrawR	withdraw remuneration
Report	1,2	identityReport	report identity disclosure
	3,4	advanceReport	report advance disclosure
	5,6	absentReport	report absent trustee
	7,8	fakeReport	report fake submission
	2,4,6,8	withdrawA	withdraw report award

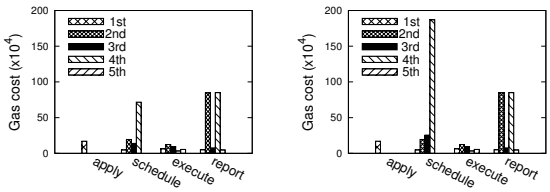
TABLE II: Key on-chain functions in solidity, the three colored functions are in proxy contract C_p, the rest of the functions are in scheduler contract C_s

shown in Table II. In both the tables, we show the components and steps where each function works in protocol. For example, function $share()$ is used in step 5 of *user schedule* component to split key to n $shares$ using Shamir secret sharing [39].

- ***Trustee application***: Any EOA in the network can invoke *newCandidate()* to join the trustee candidate pool maintained by scheduler contract C_s.
- ***User schedule***: Any EOA can invoke *newUser()* to be recorded as a user and then set up new schedule through *newSchedule()*. Then, during whisper communication with trustees, $h(onion)$ should be submitted to C_s through *setOnion()* while $h(T^{addr}, R_T)$ and $h(IN, R_U)$ should be submitted to C_s through *setTrustee()*. Meanwhile, the generation of $shares$, signatures, $onions$ and hash values are completed by *share()*, *ecsign()*, *encrypt()*, *solidity Sha3()* in node.js, respectively.
- ***Function execution***: A trustee can submit private key and $onion$ through *submitPrivkey()* and *submitOnion()*, respectively. Then, after decrypting $onions$ to $shares$ through *decrypt()* and combining $shares$ to key through *combine()*, any trustee has the ability to make the target function be executed through *execute()*. Finally, after the execution window is over, trustees can withdraw deposit and remuneration through *withdrawD()* and *withdrawR()*, respectively.
- ***Misbehavior report***: The four types of report mechanisms are implemented by *identityReport()*, *advanceReport()*, *absentReport()* and *fakeReport()*, respectively. Then, after the execution window is over, reporters can withdraw reward through *withdrawA()*.

Test instance: We design two test instances A and B:

Instance	l	m	n	5% IM
A	3	2	5	99.82%
B	4	4	10	99.95%



(a) Instance A (b) Instance B

Fig. 5: Gas cost

Instance A employs 15 trustees while instance B employs 40 trustees. As a result, instance B has higher schedule success rate under 5% inadvertent misbehaviors (IM). In both instance A and B, we use the *SealedBidAuction* contract [47] as the target contract C_t and we assumed user's goal was to schedule a transaction calling function $reveal(amount, nonce)$. Specifically, we designed an input parameter $time$ to simulate the time during testing.

B. Experimental evaluation

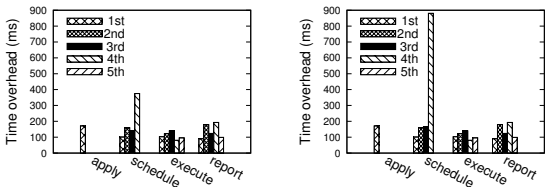
We use the presented test instances to experimentally evaluate the performance of the smart contracts, namely the gas cost and time overhead of each function presented in Table II.

Gas cost: Gas is spent in Ethereum for deploying smart contracts or calling functions. The gas costs of functions in Table II for instance A and B are shown in Figure 5(a) and Figure 5(b), respectively. For ease of presentation, results are grouped into four clusters. Each cluster represents a protocol component and contains a group of functions following their order in Table II. As can be seen, most functions cost very little. Specifically, among the fifteen functions, eight cost lower than 10^5 gas and eleven cost lower than 2×10^5 gas. Among the rest four functions, both *advanceReport()* and *fakeReport()* cost around 8.5×10^5 because the two functions need to derive public key from private key on chain. Gas costs of the last two functions, namely $setOnion()$ and $setTrustee()$, change with n and nl, respectively. From instance A to B, l increases from 3 to 4 and n increases from 5 to 10. As a result, gas cost of $setOnion()$ increases from 1.40×10^5 to 2.55×10^5 and gas cost of $settrustee()$ increases from 7.17×10^5 to 1.87×10^6.

To complete a schedule, some functions need to be invoked for multiple times. Below, we show the number of times that each function needs to be invoked in a single schedule when there is no report needed:

Function	No.	Function	No.	Function	No.
newCandidate	nl	setTrustee	1	execute	1
newSschedule	1	submitPrivkey	$n(l-1)$	withdrawD	nl
setOnion	1	submitOnion	n	withdrawR	nl

Besides, the gas cost of deploying proxy contract C_p is about 1.33×10^6. Therefore, the total gas costs of instance A and B are 7.60×10^6 and 1.72×10^7, respectively. Both gas price and Ether price keeps dramatically swinging [17].



(a) Instance A (b) Instance B

Fig. 6: Time overhead

For example, based on prices of date 12/5/2016, instance A and B cost $1.2 and $2.72, respectively. However, based on prices of date 10/29/2017, the two instances cost $22.8 and $51.6, respectively. As can be seen, the monetary cost of a timed-execution service is highly influenced by the fluctuation of cryptocurrency market, which may be a common limitation of cryptocurrency-based applications.

Time overhead: The time overheads of functions in Table II for instance A and B are shown in Figure 6(a) and Figure 6(b), respectively. All results are averaged for 100 tests. Among the fifteen functions, fourteen functions spend 0-200ms. It is the function *setTrustee()* that spends more time to record information of all the trustees to the blockchain. Specifically, *setTrustee()* spends 375ms for instance A while 881ms for instance B as there are more trustees in instance B.

VI. Related work

The problem of revealing private data at a release time in future has been researched for more than two decades. The problem was first described by May as timed-release cryptography in 1992 [29] and has intrigued many researchers since then. There are four sets of representative solutions in the literature. The first category of solutions was designed to make data recipients solve a mathematical puzzle, called time-lock puzzle, before reading the messages [9], [10], [37]. The time-lock puzzle can only be solved with sequential operations, thus making multiple computers no better than a single computer. This solution suffers from two key drawbacks. First, the time taken to solve a puzzle may be different on different computers. Second, the puzzle computation is associated with a significant computation cost, which does not lead to a scalable cost-effective solution. The second group of solutions relies on a third party, also known as a time server, to release the protected information at the release time in future. The information, sometimes called time trapdoors, can be used by recipients to decrypt the encrypted message [23], [37]. However, the time server in this model has to be trusted to not collude with recipients so that encrypted messages cannot be entered before release time. This restriction makes this set of solutions involve a single point of trust. The third set of approaches studied the problem in the context of Distributed Hash Table (DHT) networks [25], [26]. The idea behind these techniques is to leverage the scalability and distributed features of DHT P2P networks to make message securely hidden before release time. Finally, the last direction uses blockchains as a reference time clock correctness guaranteed by the distributed network [22], [28]. By combining witness encryption [19] with blockchain, one can leverage the computation power of PoW in blockchain to decrypt a message after a certain number of new blocks have been generated. However, the current implementation of witness encryption is far from practical, which requires an astronomical decryption time estimated to be 2^{100} seconds [28]. Recent work has studied the problem of supporting self-emerging data in blockchain networks [27]. It allows the encrypted private data to travel through a long path within a blockchain network and appear at the prescribed release time. However, unlike the proposed work in this paper, this approach supports only self-emergence of data and fails to support timed invocation of smart contract functions which is crucial for supporting timed executions in decentralized applications on blockchain-based smart contract platforms. Besides the above mentioned solutions, there are two tools that support timed execution of transactions, however, they do not protect sensitive inputs. *Ethereum Alarm Clock* [1] allows a client to deploy a request contract to the Ethereum network at time A with a reward and if any account is interested in the reward, the account can invoke the request contract at a prescribed Time B to make the scheduled transaction be sent to earn the reward. However, this scheme neither protects sensitive inputs nor guarantees the transaction execution. *Oraclize* [4] is a blockchain oracle service that takes the role of a trusted third party (TTP) to execute the transaction on behalf of the client at a future time point. The limitations of of this scheme include both the centralization brought by the TTP and the lack of protection of sensitive inputs. To the best of our knowledge, the approach proposed in this paper is the first decentralized solution for enabling users of decentralized applications to schedule timed execution of transactions without revealing sensitive inputs before an execution time window chosen by the users.

VII. Conclusion

In this paper, we developed a new decentralized privacy-preserving timed execution mechanism that allows users of Ethereum-based decentralized applications to schedule timed transactions without revealing sensitive inputs before an execution time window chosen by the users. The proposed approach involves no centralized party and allows users to go offline at their discretion after scheduling a timed transaction. The timed execution mechanism protects the sensitive inputs by employing a set of trustees from the decentralized blockchain network to enable the inputs to be revealed only during the execution time. We implemented the proposed approach using *Solidity* and evaluated the system on the Ethereum official test network. Our theoretical analysis and extensive experiments validate the security properties and demonstrate the low gas cost and low time overhead associated with the proposed approach.

REFERENCE

[1] Ethereum alarm clock. https://www.ethereum-alarm-clock.com/.

[2] Golem. https://golem.network/.

[3] iex.ec. https://iex.ec/.

[4] Oraclize. http://www.oraclize.it/.

[5] Marcin Andrychowicz et al. Secure multiparty computations on bitcoin. In *Security and Privacy*, pages 443–458. IEEE, 2014.

[6] Auctionhouse project. https://github.com/dob/auctionhouse.

[7] Asaph Azaria et al. Medrec: Using blockchain for medical data access and permission management. In *OBD*, pages 25–30. IEEE, 2016.

[8] Guido Bertoni et al. The keccak sha-3 submission. *Submission to NIST (Round 3)*, 6(7):16, 2011.

[9] Nir Bitansky et al. Time-lock puzzles from randomized encodings. In *ITCS*, pages 345–356. ACM, 2016.

[10] Dan Boneh and Moni Naor. Timed commitments. In *Advances in CryptologyCrypto 2000*, pages 236–254. Springer, 2000.

[11] Boomerang. https://www.boomeranggmail.com/.

[12] Roger Dingledine et al. Tor: The second-generation onion router. Technical report, Naval Research Lab Washington DC, 2004.

[13] Changyu Dong et al. Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing. *ACM CCS*, 2017.

[14] eth-ecies. https://github.com/libertylocked/eth-ecies.

[15] Ethereum market cap. https://coinmarketcap.com/currencies/ethereum/.

[16] ethereumjs-util. https://github.com/ethereumjs/ethereumjs-util.

[17] Etherscan: gas price. https://etherscan.io/chart/gasprice.

[18] Edoardo Gaetani et al. Blockchain-based database to ensure data integrity in cloud computing environments. *eprints*, 2017.

[19] Sanjam Garg et al. Witness encryption and its applications. In *STOC*, pages 467–476. ACM, 2013.

[20] Geth: Official go implementation of the ethereum protocol. https://github.com/ethereum/go-ethereum.

[21] Carmit Hazay and Yehuda Lindell. A note on the relation between the definitions of security for semi-honest and malicious adversaries. *IACR Cryptology ePrint Archive*, 2010:551, 2010.

[22] Tibor Jager. How to build time-lock encryption. *IACR Cryptology ePrint Archive*, 2015:478, 2015.

[23] Kohei Kasamatsu et al. Time-specific encryption from forward-secure encryption. In *SCN*, pages 184–204. Springer, 2012.

[24] Kristian Lauslahti et al. Smart contracts–how will blockchain technology affect contractual practices? *Discussion paper*, 2017.

[25] Chao Li and Balaji Palanisamy. Emerge: Self-emerging data release using cloud data storage. In *Cloud Computing (CLOUD), 2017 IEEE 10th International Conference on*, pages 26–33. IEEE, 2017.

[26] Chao Li and Balaji Palanisamy. Timed-release of self-emerging data using distributed hash tables. In *ICDCS*, pages 2344–2351. IEEE, 2017.

[27] Chao Li and Balaji Palanisamy. Decentralized release of self-emerging data using smart contracts. In *SRDS*. IEEE, 2018.

[28] Jia Liu et al. Time-release protocol from bitcoin and witness encryption for sat. *IACR Cryptology ePrint Archive*, 2015:482, 2015.

[29] Timothy May. Timed-release crypto. *http://www. hks. net. cpunks/cpunks-0/1560. html*, 1992.

[30] Patrick McCorry et al. A smart contract for boardroom voting with maximum voter privacy. In *FC*, pages 357–375. Springer, 2017.

[31] Andrew Miller and Iddo Bentov. Zero-collateral lotteries in bitcoin and ethereum. In *EuroS&PW*, pages 4–13. IEEE, 2017.

[32] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[33] Neo. https://neo.org/.

[34] Thanh Hong Nguyen et al. Analyzing the effectiveness of adversary modeling in security games. In *AAAI*, 2013.

[35] Postfity. https://postfity.com/.

[36] Rinkeby: Ethereum official testnet. https://www.rinkeby.io/#stats.

[37] Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. *Massachusetts Institute of Technology*, 1996.

[38] secrets.js. https://github.com/grempe/secrets.js.

[39] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[40] Smart contracts market research report global forecast to 2023. https://www.marketresearchfuture.com/reports/smart-contracts-market-4588.

[41] State of the dapps. https://www.stateofthedapps.com/.

[42] The solidity contract-oriented programming language. https://github.com/ethereum/solidity.

[43] CERT Insider Threat. Us state of cybercrime survey (2014), 2014.

[44] web3-utils. https://www.npmjs.com/package/web3-utils.

[45] Whisper. https://github.com/ethereum/wiki/wiki/Whisper.

[46] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151, 2014.

[47] Writing a sealed-bid auction contract. https://programtheblockchain.com/posts/2018/03/27/writing-a-sealed-bid-auction-contract/.