T-Watch: Towards Timed Execution of Private Transaction in Blockchains

Chao Li, *Member, IEEE,* and Balaji Palanisamy, *Member, IEEE,*

***Abstract*—In blockchains such as Bitcoin and Ethereum, transactions represent the primary mechanism that the external world can use to trigger a change of blockchain state. Timed transaction refers to a specific class of service that enables a user to schedule a transaction to change the blockchain state during a chosen future time-frame. This paper proposes T-Watch, a decentralized and cost-efficient approach for users to schedule timed execution of any type of transaction in Ethereum with privacy guarantees. T-Watch employs a novel combination of threshold secret sharing and decentralized smart contracts. To protect the private elements of a scheduled transaction from getting disclosed before the future time-frame, T-Watch maintains shares of the decryption key of the scheduled transaction using a group of executors recruited in a blockchain network before the specified future time-frame and restores the scheduled transaction at a proxy smart contract to trigger the change of blockchain state at the required time-frame. To reduce the cost of smart contract execution in T-Watch, we carefully design the proposed protocol to run in an optimistic mode by default and then switch to a pessimistic mode once misbehaviors occur. Furthermore, the protocol supports users to form service request pooling to further reduce the gas cost. We rigorously analyze the security of T-Watch and implement the protocol over the Ethereum official test network. The results demonstrate that T-Watch is more scalable compared to the state of the art and could reduce the cost by over 90% through pooling.**

***Index Terms*—Blockchain, Data Privacy, Timed Execution.**

I. Introduction

BLOCKCHAINS are distributed ledgers of transactions performed on a global state by nodes of blockchain networks. Transactions are at the heart of blockchains because they are the only things that the external world can use to trigger a change of blockchain state. In Bitcoin-like blockchains [33], transactions allow users to transfer funds among each other. In Ethereum-like blockchains [6] that support smart contracts [49], transactions also enable users to deploy and interact with smart contracts. Recent advancements in blockchain technology have led to a proliferation of transactions happening in blockchain networks. In 2021, the median number of daily Bitcoin and Ethereum transactions has grown to 270,736 and 1,245,624, respectively [10].

Timed Transaction (*TT*) refers to a class of service that enables a user to schedule a transaction to get executed to change the blockchain state during a chosen future time-frame. Many scenarios require timed transactions in blockchains. For example, just like the clients of credit card and facility companies in real-world who are usually allowed to select a predetermined time-frame during which payments are charged from their bank accounts, the clients of vendors or service providers in blockchain networks also need a way to schedule payment transactions to transfer cryptocurrencies from their blockchain accounts. For many businesses and computations running over smart contracts, the ability of *TT* to control the execution time of sensitive transactions and knowing precisely when such transactions are executed can be crucial. For instance, imagine that Alice is working on a confidential project and she needs to outsource a computation task to service providers by sending out a transaction that creates a smart contract implementing verifiable cloud computing [12], she may only want the contract to be recorded into the ledger exactly during the acceptance phase of the project because an early created contract could potentially leak her rate of progress to her competitors. Imagine another situation in which Bob attends a smart-contract-based sealed-bid auction [50] and would like his bid to get revealed via a transaction calling the *reveal()* function at the auctioneer smart contract exactly during the bid opening time-frame. Here, an early revelation could potentially leak his bid to his competitors who may have kept tracking Bob's transasctions while a late revelation will remove Bob's bid from the competition.

While there are numerous services (e.g., BlueOrion [5] and Oraclize [36]) that provide pre-scheduled execution of *TT* in blockchains, most of current implementations of *TT* are highly centralized, where users are forced to fully trust the centralized service providers, resulting in various security issues induced by a single point of trust. Moreover, even in cases where the service providers are perceived as trustworthy, the services remain vulnerable to unpredictable security breaches or internal attacks over which they have no control [9]. Recently, the emergence of blockchain technologies offers great potential to decentralize implementations of *TT*, and there have been several proposals of decentralized implementations [7], [17], [26], [27], [35]. Nevertheless, these implementations for now only support a single type of transaction in Ethereum, namely function invocation transaction. More importantly, most of the smart-contract-based protocols developed in existing decentralized approaches can hardly become practical because they typically involve $O(n)$ cost of running smart contracts among n protocol participants while they need a larger n to maintain higher security.

In this paper, we present T-Watch, a decentralized implementation of *TT* for users to schedule *any* type of transaction in Ethereum with protection of private data within scheduled transactions with significantly reduced cost of running smart contracts during the process. T-Watch employs a novel combination of threshold secret sharing and decentralized smart contracts. To protect the private elements of scheduled transactions from getting disclosed

before the future time-frame, T-Watch maintains shares of the decryption key of the scheduled transaction signed by a user using a group of executors recruited in a blockchain network before the specified future time-frame and restores the scheduled transaction at a proxy smart contract to trigger the change of blockchain state exactly during the required time-frame. To reduce the cost of running smart contracts in each timed transaction, we carefully design a protocol to run in an optimistic execution path called *T-Opt* by default, where the non-scalable regulations are cut off to reduce the cost of running smart contracts. Then, upon detecting any misbehavior, executors reserve the ability to switch the protocol to a pessimistic execution path called *T-Pes* by rebinding the removed regulations with smart contracts to redress and penalize any dishonest behavior, just as if these regulations were never decoupled. Finally, the protocol encourages users to form service request pools through an execution path called *T-Pool* so that the shares of a certain decryption key could be shared among multiple users who may have the same required time-frame. The proposed protocol incentivizes executors to stay honest and thus implements T-Watch with minimum interactions on the blockchain.

In summary, we make the following key contributions:

- To the best of our knowledge, T-Watch is the *first* practical decentralized approach designed for *TT* that is secure, scalable, privacy-preserving and cost-efficient.
- After the service has been set up, T-Watch completely isolates the service execution from the state of users, without requiring any assistance from the user side.
- We emphasize that T-Watch is a general approach that supports *all* types of transaction in Ethereum.
- T-Watch can protect the privacy of *all* elements of all types of transaction before a prescribed time-frame, including the amount of transferred funds, the content of created smart contracts, the arguments of invoked functions and even the addresses of payees or invoked smart contracts.
- We carefully design a protocol that could implement T-Watch through three different execution paths, an optimistic path *T-Opt* that spends a limited amount of gas in executing the prescribed transaction if no participant misbehaves, a pessimistic path *T-Pes* that is relatively gas-consuming but could resist misbehaviors, and a service request pooling path *T-Pool* that reduces the cost of most service requests in pools to a very small and constant value.
- We rigorously analyze the security of T-Watch against two different threat models, $\mathcal{A}$-threshold and $\mathcal{A}$-budget. We implement the protocol over the Ethereum official test network. The results demonstrate that T-Watch is more scalable compared to the state of the art and could reduce the gas cost by over 90% through pooling.

While our focus is primarily on Ethereum, the versatile design of T-Watch enables its adaptation to numerous EVM-compatible blockchains such as Binance Smart Chain, Tron, and Polygon. Additionally, by abstracting the core functionalities of T-Watch and adapting the transaction handling procedures, it can be made compatible with a variety of other blockchains that might not directly support EVM.

II. Related work

A. Timed transaction in blockchains

Existing techniques and tools for timed transaction in blockchains can be roughly divided into two categories:

Centralized approaches: Recently, many emerging blockchain services companies such as BlueOrion [5] and Oraclize [36] offer services of scheduling *TT*. These services are heavily centralized and require users to trust the companies. Alternatively, users may themselves run client-side tools such as *parity* [37] to schedule *TT*. Nevertheless, such tools typically fail to isolate users from the service execution because users have to make their machines keep connecting with the blockchain network.

Decentralized approaches: In Bitcoin, there exists a native mechanism named *Timelocks* [2] that allows users to set a timelock for a payment transaction so that the Bitcoin carried by that transaction becomes available only after the specified time point. Unfortunately, *Timelocks* is proprietary to Bitcoin and is not supported by most other blockchains such as Ethereum, so it cannot support any other type of transactions or protect sensitive elements of scheduled transactions. Recently, a project named *Ethereum Alarm Clock* [17] proposes to recruit Ethereum accounts to trigger a re-deployed contract to call a target contract during a prescribed time-frame. However, this scheme neither protects sensitive elements nor guarantees the transaction execution. A more recent work [26] further leverages threshold secret sharing [41] to offer a certain level of protection of sensitive elements, but it only supports function invocation transactions and has a cost of $O(n)$.

B. Timed release of private data

We design T-Watch to protect sensitive elements within all types of timed transactions before the prescribed time-frame, which is relevant to a classical research topic, namely the timed release of private data. The study of timed release of private data began with May [32]. Since then, there have been extensive studies on this problem, which can be roughly divided into four categories.

Mathematical puzzles: One representative approach [29], [39] protects private data with a time-lock puzzle, forcing recipients to solve a cryptographic puzzle to obtain the data. Nevertheless, the time for solving such puzzles is non-deterministic and hence, the opening time of the data can not be precisely controlled. Also, cryptographic approaches for timed data release come with a very significant computational cost and as such, these techniques are not scalable.

Time server: Another well-studied approach [16], [22] relies on a (semi-)trusted time server to release time trapdoors to recipients at specified future time points. These techniques involve a single point of trust and create a safety bottleneck.

Proof-of-Work puzzles: One direction of the decentralized approaches [28], [44] encloses private data with blockchain puzzles used in Proof-of-Work [33] and therefore minimizes the computational burden of the data recipients as the blockchain puzzles are periodically solved by blockchain miners. However in such an approach, the involved heavy

cryptographic primitives result in very high performance overhead [35], [44].

Smart contracts: Another direction of recent decentralized techniques for timed data release [7], [27], [35] leverages smart contracts to establish a decentralized autonomous agent, through which a data sender could recruit a group of peers from the blockchain network as her trustees to cooperatively maintain and deliver her private data to recipients. While the above-mentioned work considered a rational adversarial setting, the issue of protecting timed-release services in mixed adversarial environments consisting of both malicious and rational peers is addressed in [45], [47].

We emphasize the problem of timed transactions is more general and challenging than timed data release in the context of smart contracts. In timed data release, data is released to recipients who can actively leverage their accounts to interact with other participants of the protocol and exchange messages via private off-chain communication channels. In contrast, recipients of data carried by timed transactions could also be smart contract accounts that are both passive and transparent, so solutions for smart-contract-based timed data release are not capable of handling timed transactions. Besides, timed transactions enable new applications that are not supported by timed data release, including timed creation of smart contracts and timed payments of cryptocurrencies.

To sum up, our work in this paper tackles the key limitations of the state-of-the-art decentralized approaches [7], [17], [26], [27], [35]. To the best of our knowledge, *T-Watch* is the *first* practical decentralized solution for timed transaction with strong guarantees of security, scalability and cost-efficiency.

III. Preliminaries

A. Account types in Ethereum

There are two types of accounts in Ethereum:

- External Owned Account (EOA): To interact with the Ethereum ecosystem, one needs to own an EOA by locally creating a pair of asymmetric keys, leveraging the public key to generate a unique address *addr(EOA)* for this EOA and preserving the private key to sign future transactions.
- Contract Account (CA): Smart contracts in Ethereum are typically created through contract creation transactions. Each new smart contract is automatically assigned a unique address *addr(CA)* in a deterministic way.

B. Transactions and update patterns

A transaction in Ethereum is a serialized binary message sent from an EOA that contains the following key elements:

$$tx \equiv \langle to, value, data, sig \rangle \quad (1)$$

- *tx.to*: the account address of the recipient (EOA or CA).
- *tx.value*: the amount of *ether* to send to the recipient.
- *tx.data*: the binary data carried by *tx*.
- *tx.sig*: the ECDSA digital signature of the EOA.

Depending on the type of *tx.to*, transactions can be divided into three categories.

Fund transfer transaction: In Ethereum, to transfer an amount of cryptocurrency *ether* to another EOA account, one needs to create a fund transfer transaction by setting the recipient EOA as *tt.to* as well as a non-empty *tt.value*.

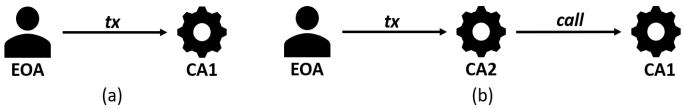


Fig. 1: EOA-only pattern (a) and EOA-CA pattern (b) to update the state of blockchain

Function invocation transaction: To invoke a function within a deployed smart contract, one needs to create a function invocation transaction by setting the CA (contract account) as *tt.to* as well as a non-empty *tt.payload*.

Contract creation transaction: To create a new smart contract, one needs to create a contract creation transaction by setting 0x0 as *tx.to* , an empty *tx.value* and a non-empty *tx.data*. Here, the created new smart contract shall be programmed using a high-level contract-oriented language such as *Solidity* [42] and then compiled into a low-level bytecode language called Ethereum Virtual Machine (EVM) code. Finally, the contract creation transaction carries the bytecode as its *tx.data*.

A user, after filling the elements, will then sign the transaction with the private key of a controlled EOA to obtain *tx.sig*. After that, following the Proof-of-Work (PoW) consensus protocol [33], [49], the created transaction is broadcasted to the Ethereum network, executed by miners and finally packed into the blockchain ledger. In Ethereum, transactions and smart contracts are executed transparently by tens of thousands of miner nodes in a decentralized manner and the results are deterministic.

Patterns to update the state of blockchain: The state of Ethereum blockchain could be updated via two patterns:

- **EOA-only pattern**: One may leverage EOAs to directly transfer funds, invoke functions or create smart contracts by establishing transactions accordingly. For instance, in Fig. 1.(a), the EOA creates a function invocation transaction to directly invoke a target function at CA1.
- **EOA-CA pattern**: Alternatively, one may leverage EOAs to create function invocation transactions only to invoke special functions at CAs, where EVM-level opcodes for system operations, such as `CALL` and `CREATE`, have been enabled. Then, the opcode `CALL` allows a contract to transfer funds to another contract or invoke a function at another contract and the opcode `CREATE` allows an existing contract to deploy a new contract. In Fig. 1.(b), the EOA creates a transaction to invoke a function at CA2, where the opcode `CALL` further triggers the invocation of the target function at CA1. Here, it is worth noting that CAs alone are unable to update the blockchain state because the execution of any function must be triggered by transactions created at EOAs from the root.

C. Gas system

It is worth noting that the execution of transactions in Ethereum costs gas, which in turn generates transaction fees. Concretely, each transaction in Ethereum first charges a basic fee of 21,000 Gas. On this basis, each instruction in the transaction execution process is charged the corresponding gas

fee according to its type [49]. Depending on the complexity of the transaction execution, the total amount of gas spent in a single transaction can be in the millions, but cannot exceed the maximum amount of gas allowed in a single block. In Ethereum, in order to send an executable transaction, one needs to ensure that the balance of the sender EOA is sufficient. This is because the *ether* corresponding to the total amount of gas spent by the transaction will be charged. The gas system is important as it helps incentivize miners to remain honest, suppress denial-of-service attacks, and encourage efficient smart contract programming. However, the gas system puts forward higher requirements for the scalability of the protocol design. In a protocol with many participants, even if the cost of a single transaction is low, the total fee of all the transactions may be very high.

D. Off-chain channels

Like most blockchain systems, nodes in Ethereum form a peer-to-peer (P2P) network. The Ethereum community proposed the Whisper protocol [48] to support inter-node communication in the P2P network. Specifically, messages sent based on the Whisper protocol are broadcast over the entire P2P network, and all messages must be encrypted symmetrically or asymmetrically and can be decrypted by the node with the corresponding key. For example, a node can generate a pair of asymmetric keys locally and store the public key on the blockchain, making the public key visible to the entire network, so as to obtain encrypted messages transmitted by other nodes using the public key and the Whisper protocol, and decrypt these messages to learn the contents. Unlike on-chain transactions that charge fees, off-chain communication costs no money.

E. Cryptographic tools

T-Watch employs several key cryptographic tools:

- **Threshold secret sharing**: A (t, n)-threshold secret sharing scheme is a method to split a secret s into n shares $\boldsymbol{s} = \{s_1, ..., s_n\}$ such that any t shares could recover the secret but $t-1$ or fewer shares fail to do that. This paper adopts the scheme proposed by Shamir based on Lagrange interpolation theorem [41], and denote the secret split and secret restoration algorithms as $\boldsymbol{s} \leftarrow SS(s, \{t, n\})$ and $s \leftarrow SR(\boldsymbol{s}, \{t, n\})$, respectively.
- **Keccak-256 hash function**: Keccak-256 [4] has been widely used in blockchain systems including Ethereum. The Ethereum Solidity language provides a function $keccak256(\cdots)\ returns\ (bytes32)$, which can directly calculate Keccak-256 hash values in smart contracts. All hash functions used in this paper are based on Keccak-256, and the hash operation is expressed as $h \leftarrow H(\cdot)$.
- **ECDSA**: Many mainstream blockchain systems such as Ethereum and Bitcoin adopt Elliptic Curve Digital Signature Algorithm (ECDSA) [21] for their public-key cryptography. Concretely, in Ethereum, one needs to first randomly generate a private key sk that satisfies the order of the secp256k1 curve and then generate the corresponding public key pk from the private key. In the rest of the paper, we simply denote the corresponding encrytion and decryption algorithms as $c \leftarrow E(pk, p)$ and $p \leftarrow D(sk, c)$, respectively. Besides, an ECDSA signature consist of two integers $\{r, s\}$, upon which Ethereum introduces an additional recovery identifier v, forming a triplet signature $\{v, r, s\}$. The Ethereum community provides a JavaScript API to sign arbitrary messages msg, and the signature operation is expressed as $vrs \leftarrow S(H(msg))$. In addition, the Ethereum Solidity language provides a function $ecrecover(\cdots)\ returns(address)$, which can directly verify the signature in smart contracts, represented as $addr(signer) \leftarrow V(H(msg), vrs)$, and the output $addr(signer)$ is the signer's EOA address.
- **ECVRF**: The Elliptic Curve Verifiable Random Function (ECVRF) [43] is a VRF that uses Elliptic Curves. Concretely, it allows a prover holding a keypair $\langle pk, sk \rangle$ from EC (e.g., secp256k1) to leverage the private key sk and an input message msg to create a pseudo-random number $r \leftarrow VR(sk, msg)$, as well as a corresponding proof $\pi \leftarrow VP(sk, msg)$. Then, the proof π allows a verifier holding the public key pk to verify that r is valid under msg and pk through $r \leftarrow VV(pk, msg, \pi)$.

IV. T-Watch: Overview

In this section, we provide an overview of T-Watch, with the intuition behind our core techniques. We first present the architectures for sending timed transactions. We then briefly depict the potential execution paths in T-Watch and introduce the core idea that reduces the on-chain cost. We defer detailed protocols fulfilling these execution paths until Section V. We finally describe the threat models in T-Watch.

A. Architectures for sending timed trasanctions

Recall that a transaction consists of the components shown in Eq. 1 and any state change must be triggered by EOAs from the root, we can abstract the following four key components from an architecture for sending timed trasanctions (tt):

$$tt \equiv \langle timer, EOA, payload, fund \rangle \tag{2}$$

- $tt.timer$: It expresses the future time-frame (e.g., start and end block numbers) scheduled for releasing tt.
- $tt.EOA$: It indicates the EOA who triggers the release of tt during the prescribed time-frame.
- $tt.payload$: It refers to the payload data carried by tt, which needs to be maintained until $tt.timer$ to be sent with tt. Here, $tt.payload$ includes the following three elements of a scheduled transaction tx:

$$tt.payload = \langle tx.to, tx.value, tx.data \rangle \tag{3}$$

- $tt.fund$: It denotes an amount of $ether$ prepared to satisfy the amount indicated by $tx.value$. Specifically, $|tt.fund| = |tx.value|$, and $tt.fund$ needs to be maintained and made available during the time-frame indicated by $tt.timer$ to be transferred with tt.

After abstracting the four key components, based on different design options for implementing them, we next present three architectures for sending timed transactions by

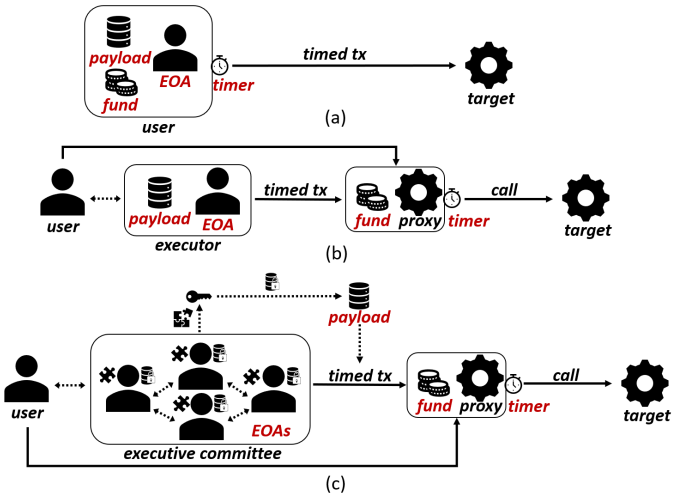


Fig. 2: The user-driven architecture (a), the executor-driven architecture (b), and the committee-driven architecture (c) for sending timed transactions. The solid/dotted lines indicate on-chain/off-chain operations, respectively.

using examples shown in Fig. 2, where the goal is to invoke a target smart contract during a prescribed time-frame.

User-driven architecture: A naive design option is to adopt the EOA-only pattern presented in Section III-B to invoke the target contract, which requires users of timed transaction service to handle all the four key components by themselves. As shown in Fig. 2.(a), in this architecture, a user needs to keep connecting with the Ethereum network, maintain $tt.payload$ locally with a timer $tt.timer$, and leverage a controlled EOA with sufficient balance to afford both $tt.EOA$ and $tt.fund$. Then, during the time-frame indicated by $tt.timer$, client tools such as *parity* [37] could help send out the scheduled transaction carrying both $tt.payload$ and $tt.fund$ from the EOA. However, the user-driven architecture cannot isolate service execution from the user side after the service has been set up, making it difficult to be adopted as a general approach.

Executor-driven architecture: To completely isolate the service execution from the user side, a natural design option is to make users recruit experienced people from the Ethereum community as executors, who may serve timed transactions with more professional equipment. Here, a naive choice is to outsource all the four key components to recruited executors so that they could simply follow the user-driven architecture on behalf of users. However, despite that the integrity of *tt.payload* could be verified using users' signatures, outsourcing *tt.fund* and *tt.timer* directly to EOAs controlled by executors is insecure because a dishonest executor may easily embezzle *ether* received from a user and may also violate the indication of *tt.timer*. Therefore, as shown in Fig. 2.(b), we adopt the EOA-CA pattern presented in Section III-B to replace the EOA-only pattern so that a *proxy* contract deployed by a user can act on behalf of the user to preserve *tt.fund* and enforce *tt.timer* with codes in a trustworthy and decentralized manner. However, the rest two components *tt.payload* and *tt.EOA* remain to be managed in a centralized way and thus become the bottleneck of the architecture because a malicious recruited executor may either disclose or abuse sensitive *tt.payload* or be derelict of its duty of being *tt.EOA*.

Committee-driven T-Watch architecture: Therefore, as shown in Fig. 2.(c), we further extend the executor-driven architecture to the committee-driven T-Watch architecture that can completely decentralize all the four components and also prevent *tt.payload* from getting disclosed before the prescribed time-frame. The key idea behind the T-Watch architecture is to recruit a group of executors to form an executive committee and jointly maintain *tt.payload* via the threshold secret sharing [41]. More specifically, a user generates a keypair $\langle pk_u, sk_u \rangle$, encrypts *tt.payload* with pk_u and splits sk_u to shares $\{s_1, ..., s_n\}$. Then, each share s_i is maintained by one or multiple executors before the specified time-frame. During the time-frame, the executors jointly restore sk_u and call the *proxy* contract with the decrypted *tt.payload*. With T-Watch, as long as a certain amount of recruited executors stay honest, *tt.payload* shall be concealed before the time-frame while revealed during the time-frame. Besides, as long as a single recruited executor is honest, the task of *tt.EOA*, namely calling the *proxy* contract shall be completed during the time-frame. Therefore, we say that this architecture is able to completely decentralize all the components and also protect the content of all the components within *tt.payload*, including *tx.to*, *tx.value* and *tx.data*. Besides, we believe that shifting the assumption from the honesty of a single executor to the honesty of some executors makes it more suitable for the blockchain context.

On/off-chain operations: The operations performed by users and executors in Fig. 2 consists of two modes widely recognized by relevant studies [23], [30].

- *On-chain operations*: As presented in Section III-B, participants can make data publicly recorded by the blockchain using function invocation transactions that carry the data in *tx.data*. We assume consistency, availability, and immutability of blockchain systems. Specifically, when a participant submits a message to the blockchain through a transaction, within a limited period of time, other participants can obtain the message from the ledger, and the content is consistent and immutable.
- *Off-chain operations*: As presented in Section III-D, participants can establish P2P off-chain channels using the Whisper protocol to deliver messages to specific recipients. We assume availability and reliability of off-chain channels. We also assume that off-chain channels conform to the synchronous model, that is, there is a definite upper bound on the delay of transmitting messages.

B. Execution paths of T-Watch

We consider three execution paths of T-Watch with different characteristics, namely the optimistic path *T-Opt*, the pessimistic path *T-Pes*, and the service request pooling path *T-Pool*:

- ***T-Opt* (optimistic)**: This optimistic path is the default option and it assumes that there would be no misbehaviors. In short, we expect *T-Opt* to incur $O(n)$ gas cost for a user to schedule a timed transaction tt by establishing a new committee and only $O(1)$ gas cost for the committee to execute the prescribed tt if no participant misbehaves.

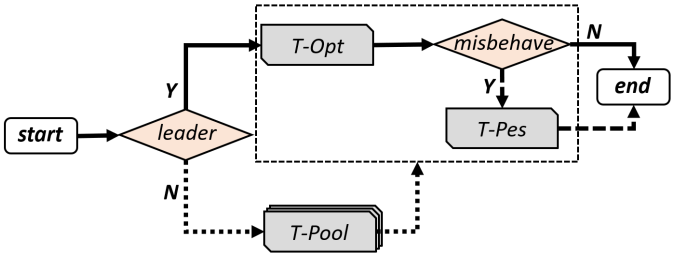


Fig. 3: A flowchart of the three execution paths of T-Watch.

- ***T-Pes* (pessimistic)**: As the back-up option, the *T-Pes* path is pessimistic, which is gas-consuming but could resist misbehaviors. The path is designed to be activated to replace *T-Opt* only at the moment when misbehaviors occur. It could resist misbehaviors at the expense of $O(n)$ gas cost for the committee to execute the prescribed timed transaction tt.
- ***T-Pool* (service request pooling)**: For a user (say Bob) who may not want to afford the $O(n)$ gas cost of setting up a committee, *T-Pool* allows Bob to join a service request pool as a follower. The leader of the service request pool is another user (say Alice), whose committee would later execute Bob's tt at the same time-frame prescribed by Alice along with her tt, thus reducing Bob's cost from $O(n)$ to $O(1)$. In each service request pool, all the followers directly inherit the same level of resistance towards misbehaviors from the leader. Therefore, *T-Pool* also relies on *T-Pes* to handle potential misbehaviors.

Next, we present the strategy for selecting and switching execution paths using a flowchart shown in Fig. 3. At the outset, a user needs to make a choice between *T-Opt* and *T-Pool*. Here, a user employing *T-Opt* would become a leader who initializes a new service request pool of tt while a user who has chosen *T-Pool* would become a follower to join an existing pool. It is worth noting that a follower should only join a service request pool if the future time-frame prescribed by the leader meets the demand of the follower. After that, in *T-Opt*, if no participant misbehaves before the prescribed time-frame, the timed transaction tt will be released during the time-frame and the entire process will complete with *T-Opt*. Otherwise, *T-Opt* will be switched to *T-Pes* so that misbehaviors can be appropriately addressed and the process will complete with *T-Pes*. Besides, in *T-Pool*, depending on whether misbehaviors occur or not, *T-Pool* will be switched to *T-Pes* or *T-Opt*, respectively, and the process will complete with *T-Pes* or *T-Opt*, correspondingly.

C. Threat Model and Assumptions

Recent works [12], [25] on blockchain-based protocol design suggest four different types of participants of a protocol, namely honest (never violate a protocol), semi-honest, malicious and rational participants:

- ***Semi-honest participants***: They behave according to the protocol but try to obtain more information from the available intermediate results.
- ***Malicious participants***: They can take any malicious actions without regard to their own interests.
- ***Rational participants***: The behavior of this type of participant is driven by self-interest. When the expected payoff from following a protocol is higher, the protocol will be followed, otherwise, the protocol will be violated.

Recent studies [20], [34] believe that in many practical situations, the assumption of semi-honest participants is too weak, while that of malicious participants is too strong. Therefore, the assumption of rational participants driven by personal interests is considered to be more realistic in many attack scenarios in blockchain [11], [12], [24], [25].

In this paper, we assume that there exists an adversary $\mathcal{A}$ seeking the premature disclosure of private *tt.payload* before the future time-frame. For instance, Bob's competitors in a sealed-bid auction may want to obtain the *tt.payload* from Bob's tt to learn Bob's bid before the bid opening time-frame. It is easy to see that such premature disclosure could happen only when $\mathcal{A}$ acquires at least t shares out of a total of n shares of the decryption key before the time-frame.

We present two attack methods that $\mathcal{A}$ may mount to disclose the protected *tt.payload* before the prescribed time-frame in the context of T-Watch:

- ***Sybil attack***: With Sybil attacks [13], $\mathcal{A}$ can create a large number of EOAs and use them to occupy as many as positions of recruited executors of a targeted user. Consequently, immediately after the shares are assigned to the recruited executors, it is possible for $\mathcal{A}$ to restore the decryption key and obtain *tt.payload* in plaintext.
- ***Bribery attack***: Instead of occupying the positions of recruited executors in person, $\mathcal{A}$ may choose to prepare a fund and use the fund as a reward to bribe executors recruited by a targeted user that are not controlled by $\mathcal{A}$. In addition, $\mathcal{A}$ could even create a bribery smart contract to establish fair collusion with a disloyal executor recruited by the targeted user [12].

We understand that it is difficult to completely prevent these attacks in public blockchains, especially when $\mathcal{A}$ has unlimited power. This paper considers two different threat models:

$\mathcal{A}$-threshold: Inspired by the threat models adopted in [30], [51], $\mathcal{A}$-threshold assumes that $\mathcal{A}$ is malicious, but could corrupt no more than $tl-1$ executors through either Sybil or Bribery attack in a committee consisting of nl executors, where t shares are generated through (t,n)-threshold secret sharing and each share is jointly maintained by l executors. Besides, $\mathcal{A}$-threshold assumes that users and the remaining executors are honest. Under this model, we define a $\mathcal{A}$-threshold-resistant protocol as follows:

Definition 1. *A $\mathcal{A}$-threshold-resistant protocol satisfies the secrecy property:*
***Secrecy**: If $\mathcal{A}$ corrupts no more than $tl-1$ executors in the committee, $\mathcal{A}$ learns no extra information about sk_u.*

$\mathcal{A}$-budget: Recent efforts [12], [31] frequently leverage cryptocurrency (e.g., *bitcoin*, *ether*) as security deposits to incentivize participants to obey protocols. Inspired by them, $\mathcal{A}$-threshold assumes that $\mathcal{A}$ is malicious, but has a bounded attack budget denoted as b, namely a bounded total amount of *ether* that can be invested in either Sybil or Bribery

notation	description
U_l	a user (leader) of Timed Transaction (TT)
U_f	a user (follower) of Timed Transaction (TT)
E	an executor in TT
$\boldsymbol{E}$	an execurive committee in TT
C	a smart contract
$C.fun()$	function $fun()$ within contract C
$\rightrightarrows$	broadcast information via off-chain channels
$\dashrightarrow$	transmit infomation via private off-chain channels
$\Rightarrow$	invoke a function within a contract
$addr(*)$	an address of an EOA or a CA

TABLE I: Summary of notations.

attack. Besides, $\mathcal{A}$-threshold assumes that all the participants, including executors and users, are rational. Under this model, we denote the prescribed amount of security deposit per service per executor as Δd and define a $\mathcal{A}$-budget-resistant protocol as follows:

Definition 2. *A $\mathcal{A}$-budget-resistant protocol satisfies the following properties:*
***Sybil-resistance**: There exists a lower bound on the attack budget of acquiring an expected value of t shares in a Sybil attack, which offers linear scaling of Δd with the number of executors independent of $\mathcal{A}$.*
***Bribery-resistance**: There exists a lower bound on the attack budget of acquiring t shares in a bribery attack, which offers linear scaling of Δd with tl.*

V. T-Watch: Protocol

In this section, we start by providing an overview of the protocol that implements T-Watch. We then identify essential requirements and describe a number of key design options. Finally, we depict the proposed protocol in detail. Throughout Section V, we assume that a user is scheduling a function invocation transaction to call a function within a target smart contract denoted as C_t. We summarize the notations that will be used in this section in TABLE I.

A. Protocol overview

In Fig. 4, we sketch the overall protocol as a three-phase process for serving *TT* in the context of the committee-driven T-Watch architecture: (1) *TT.schedule*, a user (leader) establishes a new committee during this phase; (2) *TT.waiting*, the committee preserves the encrypted *tt.payload* and shares during this phase; (3) *TT.execute*, the committee executes the timed transaction during this phase. Next, following the order of these three phases, we provide an overview of the protocol under *T-Opt*, *T-Pes* and *T-Pool*, respectively.

T-Opt: During *TT.schedule*, a user who decides to become a leader (U_l) and establish a new committee needs to sequentially perform three operations. Concretely, the leader first deploys a proxy contract C_p with *tt.fund* (if needed) and *tt.timer*, then announces her service requirements and declares the list of a new committee ($\boldsymbol{E}$) at a bulletin-board smart contract denoted as C_b, and finally delivers her encrypted private data to $\boldsymbol{E}$ via off-chain channels. After that, no action would be required until *TT.execute*. During the prescribed time-frame, all the executors within $\boldsymbol{E}$ need to reveal their service private keys sk_e (not their account private keys) via off-chain channels so that they can sequentially decrypt shares, restore sk_u, decrypt *tt.payload* and finally execute the timed transaction to call the target contract C_t via C_p.

T-Pes: *T-Opt* expects executors to honestly reveal their service private key sk_e during the prescribed time-frame. However, in practice, executors may choose to reveal sk_e before the prescribed time-frame, never reveal sk_e, or reveal fake sk_e. Despite the use of the (t, n)-threshold secret sharing, corrupted sk_e may result in more than $n - t$ unavailable shares. Therefore, we design *T-Pes* to replace *T-Opt*, when *T-Opt* has failed to restore sk_u during the first epoch of *TT.execute*. Specifically, after confirming that *T-Opt* has failed, an executor could deploy a prescribed supplemental contract C_s from the proxy contract C_p and become a watchdog. Then, executors are incentivized to send real sk_e to C_s within a short time window because any leaked, missing or fake sk_e would be confirmed and penalized after this time window.

T-Pool: A follower U_f could join a service request pool associated with U_l and $\boldsymbol{E}$ during *TT.waiting*. Here, U_f needs to deliver the private data to $\boldsymbol{E}$ via off-chain channels. The private data of U_f shall be encrypted by the service public key pk_u of U_l who has established $\boldsymbol{E}$ during *TT.schedule*.

B. Key design options

Punishment mechanism: To resolve the problem of lacking a way of pushing executors to behave honestly, we propose to design the protocol with a punishment mechanism that penalizes dishonest executors by confiscating their security deposits and rewards reporters of misbehaviors using confiscated security deposits. The idea of employing such a punishment mechanism is inspired by recent efforts that leverage cryptocurrency as deposits to improve security, including using bitcoin to penalize anyone who unfairly aborts a secure multiparty computation (SMC) [1], as well as using *ether* as security deposits to provide verifiable cloud computing [12] or to enforce certificate authorities to be honest [31].

Reputation-weighted remuneration mechanism: To attract high-quality executors and build a sustainable T-Watch ecosystem, we propose a reputation-weighted remuneration mechanism. Initially, an executor has a starting reputation score of $r = r_l$ and a starting difficulty coefficient of $\tau = 1$. Then, every τ times this executor successfully completes a T-Watch task, the reputation score increases by a step length of Δr until it reaches a prescribed upper bound $r = r_u$, and the value of τ also increases by one to enhance the difficulty of improving reputation. Meanwhile, after each task, the execution receives remuneration of $r\Delta p$ or $r\Delta p + \zeta$, where Δp denotes the amount of remuneration per unit of reputation score, and ζ denotes a bonus paid to the executor who invokes the execution function at C_p, as shown in Fig. 4. However, upon being convicted of any misbehavior, the executor will be immediately blacklisted and lose the score. The reputation-weighted remuneration mechanism helps further increase the implicit cost of misbehaving, which will be presented with more details in Section VI-B.

Splitting complicated smart contracts: To reduce the gas cost of deploying a complicated smart contract that

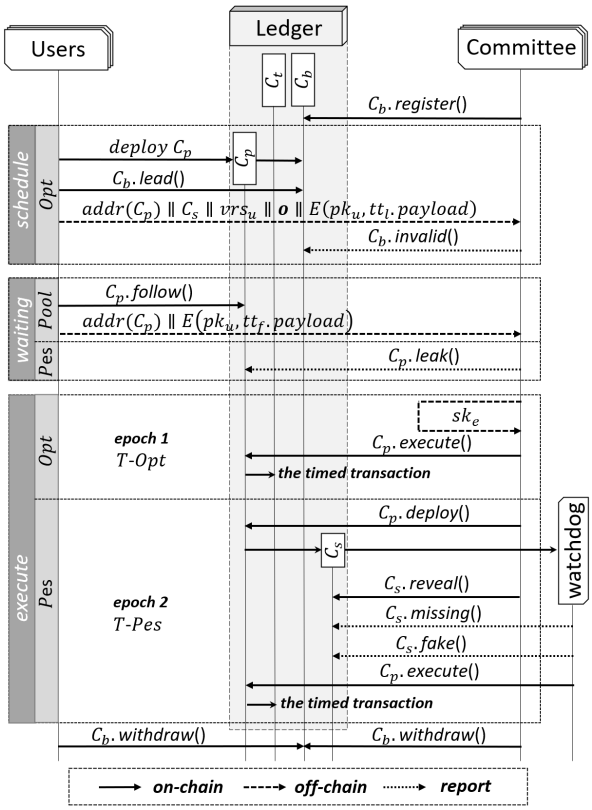


Fig. 4: An overview of the proposed T-Watch protocol.

unnecessarily includes all the functions that support all the three paths, we propose to divide such a complex contract into two separate contracts, namely a proxy contract C_p that always needs to be deployed, as well as a supplemental contract C_s that only needs to be conditionally deployed. By default, a user deploys C_p during *TT.schedule*, which then serves both *T-Opt* and *T-Pool*. If needed, an executor deploys C_s to rebind it with C_p during the second epoch of *TT.execute*, which later supports *T-Pes*.

C. Protocol detailed description

We show the detailed protocol in Fig. 5. Before *TT.schedule*, it is worth noting that the protocol demands each executor to register with the bulletin-board contract C_b through $register()$. Concretely, an exexutor E_i needs to provide three things to C_b:

- A public key for facilitating off-chain communications through the Whisper protocol introduced in Section III-D;
- An amount of $k_i \Delta d$ *ether* as a security deposit, where Δd denotes the prescribed amount of deposit per service and k_i denotes the maximum number of services E_i intends to simultaneously participate in;
- A total of m_i independent public service keys, each of which is randomly created by E_i in the form of $\langle pk_i^e, sk_i^e \rangle$ for attending a different service, and the corresponding private service keys must be well maintained by E_i.

Then, every time E_i gets selected for an executive committee to serve a user (leader), one service keypair $\langle pk_i^e, sk_i^e \rangle$ needs to be consumed and deleted. Consequently, the number of leaders that E_i can simultaneously serve is $min(k_i, \hat{m}_i)$,

TT.schedule [Opt]:

1. A user (leader) U_l creates C_p and C_s locally, deploys C_p with $tt_l.fund$. The deployed C_p immediately informs C_b of both the current block number bn and $addr(C_p)$. Meanwhile, U_l fetchs the full list of registered executors $\boldsymbol{R}$ at bn from C_b.
2. U_l generates a service keypair $\langle pk_u, sk_u \rangle$, creates a pseudo-random number $r \leftarrow VH(sk_u, addr(C_b) \parallel bn)$ and a proof $\pi \leftarrow VP(sk_u, addr(C_b) \parallel bn)$. Then, $\forall i \in \{1, ..., nl\}$, U_l selects E_i for the committee $\boldsymbol{E}$ by picking the first available executor starting from $R_{H(r,i)\%|\boldsymbol{R}|}$ in $\boldsymbol{R}$. Finally, $U_l \Rightarrow C_b.lead(tt_l.timer, pk_u, \{\pi, r, bn\}, \{l, t, n\}, addr(C_p), \boldsymbol{E})$.
3. U_l obtains shares $\boldsymbol{s} \leftarrow SS.split(sk_u, \{t, n\})$, and encrypts shares $\boldsymbol{s}$ to onions $\boldsymbol{o}$ using public service keys of $\boldsymbol{E}$, namely $\forall i \in \{1, ..., |\boldsymbol{E}|\}$, $o_i \leftarrow E(pk^e_{l(i-1)+l}, ..., E(pk^e_{l(i-1)+1}, s_i)...)$.
4. $\forall i \in \{1, ..., |\boldsymbol{E}|\}$, $U_l \dashrightarrow E_i$: $addr(C_p) \parallel C_s \parallel vrs_u \parallel \boldsymbol{o} \parallel E(pk_u, tt_l.payload)$, where $vrs_u \leftarrow S(H(addr(C_p) \parallel C_s))$.
5. Upon detecting any invalid $E_j \in \boldsymbol{E}$ uploaded in step 2, any $E_i \in \boldsymbol{E}$ could report it through $E_i \Rightarrow C_b.invalid(j)$.

TT.waiting [Pool]:

6. Each user (follower) $U_f \Rightarrow C_p.follow(tt_f.fund)$.
7. $\forall i \in \{1, ..., |\boldsymbol{E}|\}$, $U_f \dashrightarrow E_i$: $addr(C_p) \parallel E(pk_u, tt_f.payload)$.

TT.waiting [Pes]:

8. Upon detecting leakage of any $sk_i^e \in \{sk_1^e, ..., sk_{nl}^e\}$, any account $\Rightarrow C_p.leak(sk_i^e, pk_i^e)$.

TT.execute.epoch-1 [Opt]:

9. $\forall i \in \{1, ..., |\boldsymbol{E}|\}$, $E_i \dashrightarrow \boldsymbol{E} : addr(C_p) \parallel sk_i^e$.
10. $\forall i$, E_i restores $s_i \leftarrow D(sk^e_{l(i-1)+1}, ..., D(sk^e_{l(i-1)+l}, o_i)...)$, and the successfully restored s_i forms $\hat{\boldsymbol{s}}$.
11. If $|\hat{\boldsymbol{s}}| \geq t$, any E_i does the following and the protocol completes with state `SUCCESS`.
    - 11.1. Restore $sk_u \leftarrow SR(\hat{\boldsymbol{s}}, \{t, n\})$ and obtain $tt_l.payload \leftarrow D(sk_u, E(pk_u, tt_l.payload))$.
    - 11.2. $E_i \Rightarrow C_p.execute(tt_l.payload)$.
    - 11.3. Repeat step 11.1 and step 11.2 for all $tt_f.payload$ received in step 7.

    Otherwise, the protocol goes to epoch 2 (*T-Pes*).

TT.execute.epoch-2 [Pes]:

12. Any $E_i \in \boldsymbol{E}$ could deploy C_s through C_p via $E_i \Rightarrow C_p.deploy(addr(C_p), C_s, vrs_u)$, thus turning the protocol from *T-Opt* into *T-Pes*. The executor who deploys C_s then becomes a watchdog W.
13. $\forall i \in \{1, ..., |\boldsymbol{E}|\}$, $E_i \Rightarrow C_s.reveal(sk_i^e)$, and the correctly uploaded s_i forms $\hat{\boldsymbol{s}}$.
14. Upon detecting any missing sk_i^e in step 13, W could report this misbehavior through $W \Rightarrow C_s.missing(addr(E_i))$.
15. Upon detecting any fake sk_i^e in step 13, W could report this misbehavior through $W \Rightarrow C_s.fake(addr(E_i))$.
16. If $|\hat{\boldsymbol{s}}| \geq t$, W does the following and the protocol completes with state `SUCCESS`.
    - 16.1. Restore $sk_u \leftarrow SR(\hat{\boldsymbol{s}}, \{t, n\})$ and obtain $tt_l.payload \leftarrow D(sk_u, E(pk_u, tt_l.payload))$.
    - 16.2. $W \Rightarrow C_p.execute(tt_l.payload)$.
    - 16.3. Repeat step 14.1 and step 14.2 for all $tt_f.payload$ received in step 7.

    Otherwise, the protocol completes with state `FAILURE`.

Fig. 5: The protocol that implements T-Watch. A step with a gray bullet (e.g., 3) refers to an off-chain operation not recorded by blockchain while a step with a white bullet (e.g., 2) refers to an on-chain operation recorded by blockchain.

namely the smaller one between k_i and $\hat{m}_i$, where $\hat{m}_i$ denotes the remaining public service keys recorded in C_b. Besides, E_i can adjust k_i by either transferring more *ether* to C_b or withdrawing unlocked *ether* from C_b, and should supplement new public service keys in a timely manner.

TT.schedule **[Opt]**: In the first step, a leader U_l needs to create two contracts C_p and C_s locally. Specifically, C_p consists of three execution functions and a few utility functions. The execution functions are developed to process the three categories of transactions presented in Section III-B, respectively, by first verifying the four key components shown in Eq. 2, and then employing EVM-level opcodes `CALL` and `CREATE`. We denote the three execution functions as a single function named *execute()* in the rest of this section. Besides, a typical utility function is *deploy()*, through which any executor E_i can turn the protocol from *T-Opt* into *T-Pes* by deploying C_s that contains additional utility functions such as *reveal()*, *leak()*, *missing()* and *fake()*. After creating the two contracts, U_l needs to deploy C_p, which immediately notifies both the current block number bn and the address $addr(C_p)$ to C_b. Meanwhile, U_l needs to fetch the full list of registered executors $\boldsymbol{R}$ and their status of availability at bn from C_b.

In step 2, U_l needs to first create an independent service keypair $\langle pk_u, sk_u \rangle$, where pk_u would later be used by both U_l and potential followers U_f to encrypt their private data sent to $\boldsymbol{E}$. After that, U_l could obtain a verifiable pseudo-random number $r \leftarrow VH(sk_u, addr(C_b) \parallel bn)$ and a corresponding proof $\pi \leftarrow VP(sk_u, addr(C_b) \parallel bn)$, where $addr(C_b) \parallel bn$ is used as the input message to make r unique to U_l at bn. To construct an executive committee $\boldsymbol{E}$, U_l needs to select the ith member E_i of $\boldsymbol{E}$ by pseudo-randomly finding a registered executor that has its index in $\boldsymbol{R}$ to be $H(r,i)\%|\boldsymbol{R}|$ and then picking the first available executor starting from $R_{H(r,i)\%|\boldsymbol{R}|}$ in $\boldsymbol{R}$ as E_i. In this way, given r and bn, $\boldsymbol{E}$ is deterministic and thus verifiable. Finally, U_l needs to inform C_b about the detailed information, including an expected time-frame $tt_l.timer$, the public service key pk_u, $\{\pi, r, bn\}$ required for proving $\boldsymbol{E}$, three parameters $\{l, t, n\}$, $addr(C_p)$ and $\boldsymbol{E}$, and finally an amount of *ether* consisting of a security deposit to store in C_b and a remuneration to pay $\boldsymbol{E}$.

After that, in step 3, U_l needs to further split sk_u into n shares $\boldsymbol{s} = \{s_1, ..., s_n\}$ based on (t, n)-threshold secret sharing. Instead of directly sending shares $\boldsymbol{s}$ to executors $\boldsymbol{E}$, each share s_i needs to be iteratively encrypted with l public service keys belonging to different executors, which thus makes the size of $\boldsymbol{E}$ nl, as presented in step 2. In this way, each share s_i could be turned into an onion o_i and its recovery needs the corresponding l private service keys maintained by the same set of executors. This design helps turn the leakage of a share s_i into the leakage of l private service keys so that the corresponding executors could be held accountable because each leaked sk_i^e could be verified using a pk_i^e stored in C_b and sk_i^e is known only by E_i. We will later discuss how this design can help make the protocol resilient against Sybil attacks in Section VI.

Finally, in step 4, through private off-chain channels, U_l transmits $tt_l.payload$ encrypted by pk_u to $\boldsymbol{E}$, along with $addr(C_p) \parallel C_s$ and a corresponding signature vrs_u, as well as all the onions $\boldsymbol{o}$. In step 5, given pk_u and $\{\pi, r, bn\}$, any E_i could first verify r through $r \leftarrow VV(pk_u, addr(C_b) \parallel bn, \pi)$ and then verify $\boldsymbol{E}$ using r and bn. If either r or $\boldsymbol{E}$ is invalid, E_i could call *invalid()* to make C_b verify r and a chosen E_j in $\boldsymbol{E}$. In case either r or E_j is proved to be invalid by C_b, the security deposit of U_l would be transferred to E_i, the service would be canceled, and U_l would be blacklisted.

TT.waiting **[Pool]**: During *TT.waiting*, a follower U_f could reuse the already deployed C_p and $\boldsymbol{E}$ to schedule a transaction to be executed during the same time-frame prescribed by U_l. Specifically, in step 6, U_f needs to first inform C_p about a new follower by calling a utility function named *follow()* and sending $tt_f.payload$ to C_p. Besides, U_f also needs to transfer a small amount of *ether* to C_p to pay the executor who will later trigger C_p to serve U_f, and also the leader U_l for the efforts of establishing C_p and $\boldsymbol{E}$. After that, in step 7, U_f needs to encrypt $tt_l.payload$ using pk_u and transmit the encrypted payload data to $\boldsymbol{E}$. It is worth noting that, by default, no restrictions are placed upon the maximum number of followers of a leader, but it is easy for leaders to set a restriction through C_p.

TT.waiting **[Pes]**: Meanwhile, during *TT.waiting*, the protocol stipulates that executors need to protect their private service keys well, but it is possible that some executors disclose their keys to earn profit. Due to the various online or offline ways of disclosing a sk_i^e, it is difficult to proactively detect leakage, but we could incentive anyone who deliberately or accidentally finds out a sk_i^e to report it to C_p. Thus, step 8 indicates that, during *TT.waiting*, any account in Ethereum could report a leaked sk_i^e by calling a reporting function named *leak()* and indicating the corresponding pk_i^e. If the leakage is confirmed by C_p, the security deposit of the executor that owns the leaked sk_i^e will be confiscated and further split into two parts, a larger part that rewards the reporter, as well as a smaller part that rewards U_l. The protocol splits the reward to discourage executors from intentionally reporting themselves to withdraw their security deposits.

TT.execute.epoch-1 **[Opt]**: The *TT.execute* phase consists of two epochs, which corresponds to *T-Opt* and *T-Pes*, respectively. At *epoch-1*, namely *T-Opt*, executors are required to reveal their private service keys to each other via off-chain channels (step 9), and then try to decrypt and restore as many shares as possible (step 10). After that, in step 11, depending on the number of successfully restored shares, the protocol branches out into two directions. If there are more than t available shares, executors will be able to recover sk_u, decrypt both $tt_l.payload$ and $tt_f, payload$, and call the execution function *execute()*, which will make the protocol complete with state `SUCCESS`. Otherwise, the protocol could not come to state `SUCCESS` at the end of *TT.execute*, and it will have to enter *epoch-2*.

TT.execute.epoch-2 **[Pes]**: In step 12, any executor could deploy C_s by invoking the utility function *deploy()* at C_p, which turns the protocol from *T-Opt* into *T-Pes*. The deployer of C_s automatically becomes a watchdog denoted as W, whose invocations of execution or reporting functions in the remaining steps will be prioritized, while invocations made by other executors must wait for a short period of time assigned to

W. Then, in step 13, within a short time window, executors in $\boldsymbol{E}$ need to reveal their private service keys to C_s using the function *reveal()*. If there is any missing or fake sk_e, W could report it by calling the reporting functions *missing()* and *fake()*, in step 14 and 15, respectively. Finally, similar to step 10, depending on the number of available shares, the protocol branches out into two final states in step 16, namely `SUCCESS` and `FAILURE`. Besides, the protocol offers a function named *withdraw()* that allows participants to withdraw their unlocked *ether* from C_b at any time.

VI. Security analysis

In this section, we analyze the security of the proposed T-Watch protocol regarding the two threat models, $\mathcal{A}$-threshold and $\mathcal{A}$-budget, introduced in Section IV-C.

A. Security against $\mathcal{A}$-threshold

Theorem 1. *Protocol T-Watch is $\mathcal{A}$-threshold-resistant by Definition 1.*

Recall that Definition 1 requires a secure protocol to satisfy the secrecy property. We prove it in Lemma 1.

Lemma 1. *Protocol T-Watch satisfies secrecy.*

Proof. As presented in Fig. 5, the protocol contains eight steps before *TT.execute*. Besides the registration information of executors, the information acquired by the adversary $\mathcal{A}$, who is able to corrupt up to $tl-1$ executors in the executive committee $\boldsymbol{E}$, includes:

- Before step 1: for each E_i, $addr(E_i)$ and pk_i^e; in case of a corrupted E_i, sk_i^e as well;
- Step 1: $addr(U_l)$, $addr(C_p)$, C_p, $tt_l.fund$;
- Step 2: pk_u, $\{\pi, r, bn\}$, $\{l, t, n\}$, $\boldsymbol{E}$, $tt_l.timer$;
- Step 4: C_s, $E(pk_u, tt_l.payload)$, vrs_u, $\boldsymbol{o}$;
- Step 6: for each U_f, $addr(U_f)$ and $tt_f.fund$;
- Step 7: for each U_f, $E(pk_u, tt_f.payload)$.

The information above suggests that $\mathcal{A}$ could learn pk_i^e for each E_i before step 1, pk_u, $\{\pi, r, bn\}$ and $\{l, t, n\}$ in step 2, $E(pk_u, tt_l.payload)$, vrs_u and $\boldsymbol{o}$ in step 4, and $E(pk_u, tt_f.payload)$ in step 6. Besides, in the worst case, $\mathcal{A}$ corrupts $tl-1$ executors in $\boldsymbol{E}$ and obtains their private service keys. The remaining information is obviously independent of sk_u. Next, we complete the proof as follows:

- First, based on the assumption of the hardness of the Elliptic Curve Discrete Logarithm problem (ECDLP) underpinning ECDSA, $\mathcal{A}$ learns no extra information about sk_u from pk_u, $E(pk_u, tt_l.payload)$ and $E(pk_u, tt_f.payload)$, and learns no extra information about sk_e from pk_e and $\boldsymbol{o}$.
- Then, based on the assumption of the hardness of the Decisional Diffie-Hellman (DDH) problem underpinning ECVRF, $\mathcal{A}$ learns no extra information about sk_u from $\{\pi, r, bn\}$.
- Finally, given $\{l, t, n\}$ and $tl-1$ executors' private service keys, in the worst case, $\mathcal{A}$ could obtain $t-1$ shares of sk_u. Due to the information theoretic security of Shamir's secret sharing scheme, $\mathcal{A}$ learns no extra information about sk_u from the obtained $t-1$ shares.

Therefore, protocol T-Watch satisfies secrecy. □

B. Security against $\mathcal{A}$-budget

Theorem 2. *Protocol T-Watch is $\mathcal{A}$-budget-resistant by Definition 2.*

Recall that Definition 2 requires a secure protocol to satisfy Sybil-resistance and bribery-resistance. We prove the two properties sequentially.

Lemma 2. *Protocol T-Watch satisfies Sybil-resistance.*

Proof. The adoption of ECVRF ensures both uniqueness and pseudorandomness of r, which is then used in constructing $\boldsymbol{E}$. Next, we complete the proof as follows:

- We divide all the registered executors (not just the ones in $\boldsymbol{E}$) into two groups, a group of g_s executors controlled by $\mathcal{A}$, and a group of g_o executors that are independent of $\mathcal{A}$.
- Then, to acquire a certain share, $\mathcal{A}$ needs to control the l corresponding executors all together to get their private service keys, which gives a probability of $p = (\frac{g_s}{g_s+g_o})^l$.
- Given a total of n shares, by modeling the problem as a binomial distribution with n and p, the attack budget of acquiring an expected value of t shares is $b = \frac{tg_s\Delta d}{np}$.
- To compute the lower bound of b, we make $\frac{\partial b}{\partial g_s} = 0$:

$$\begin{aligned} l(g_s+g_o)^{l-1}x^{l-1} &= (l-1)(g_s+g_o)^l x^{l-2} \\ lg_s &= (l-1)(g_s+g_o) \\ g_s &= (l-1)g_o \end{aligned}$$

Thus, b is bounded by $(l-1)g_o\Delta d$, which proves that protocol T-Watch satisfies Sybil-resistance. □

Lemma 3. *Protocol T-Watch satisfies bribery-resistance.*

Proof. Due to the mutual distrust between $\mathcal{A}$ and an independent executor, during bribery, the executor would assume that $\mathcal{A}$ will report the leakage to C_p, which will potentially make the executor get blacklisted and lose both reputation score and security deposit. Thus, to bribe a rational executor, $\mathcal{A}$ has to pay the executor an amount of *ether* higher than the executor's potential loss, which consists of two parts:

- Punishment mechanism: an amount of Δd due to the loss of security deposit;
- Reputation-weighted remuneration mechanism: an amount of $\sum_{i=0}^{\frac{r-r_l}{\Delta r}} i(\frac{r-r_l}{\Delta r} - i + 1)\Delta r\Delta p$ due to the lose of remuneration during the reconstruction of a reputation score of r from the starting score of r_l by registering a new executor.

Thus, b is bounded by $\sum_{j=0}^{tl}(\Delta d + \sum_{i=0}^{\frac{r_j-r_l}{\Delta r}} i(\frac{r_j-r_l}{\Delta r} - i + 1)\Delta r\Delta p)$, which proves that protocol T-Watch satisfies bribery-resistance. □

Remark: From the analysis above, we could observe that the security deposit Δd introduced in the punishment mechanism plays a key role in the budget b in both Sybil and bribery attacks. Besides, we could see that b in a Sybil attack is independent of both the threshold t and the size g_s of the group of executors controlled by $\mathcal{A}$, but tends to be proportional to l and g_o, which suggests that users may dynamically adjust l to customize the resistance of their services against Sybil attacks, and may choose to set a relatively smaller l when the

scale of registered executors is large. Finally, we could see that the reputation-weighted remuneration mechanism helps further increase b because of the additional compensation paid to betrayers, especially those with high reputation scores.

C. Discussion

Security against rational or malicious users: Recall that, to discourage executors from intentionally reporting themselves, the protocol splits confiscated security deposits into two parts and rewards both reporters and users. This design, however, may incentivize rational or malicious users to cooperate with $\mathcal{A}$ to defraud honest executors. In the worst case, a user (leader) may choose to become $\mathcal{A}$. Thanks to the careful design of onions, our protocol intentionally prevents any executor from obtaining the plaintext of any share because shares are also known by leaders. Instead, executors would be panelized only when their private service keys are leaked, missing or fake. In addition, executors are required to create independent keypairs for different services, so keys for a certain service would be random and independent of keys for other services. Consequently, based on the assumption of the hardness of ECDLP, our protocol can protect honest executors from getting defrauded by rational or malicious users.

Security of followers' tt: Recall that in the *T-Pool* path, a follower U_f needs to rely on an existing committee $\boldsymbol{E}$ established by a leader U_l. Therefore, the security of followers' tt relies on the pseudorandomness of $\boldsymbol{E}$. In the worst case that U_l is $\mathcal{A}$, due to the use of ECVRF, the pseudorandomness of r could be guaranteed. Then, based on the proof of Lemma 2, b in a Sybil attack is independent of the threshold t, which suggests that the strategy of repeatly recomputing r to make more controlled executors get involved in $\boldsymbol{E}$ is not helpful. In addition, the payments made by followers to leaders in step 5 of the proposed protocol would incentivize leaders to select executors in a more transparent and trustworthy manner so that they could attract more followers and earn more profit. Besides, as long as there is a single rational executor in $\boldsymbol{E}$, followers' timed transactions would be executed as expected.

Rationality of participants: The underlying assumption in our threat model $\mathcal{A}$-budget, similar to recent works [12], [19], [31], [46] , is that participants are rational. This assumption is prevalent in current blockchain and cryptographic research and posits that participants will avoid actions that result in a net loss, such as forfeiting a security deposit. While this approach is effective in mitigating most forms of misbehavior among rational actors, it is acknowledged that it may not deter all types of misbehavior, especially those conducted by irrational or extremely adversarial participants. As such, our future work will focus on enhancing the robustness of our mechanisms to include strategies that can handle scenarios involving irrational or non-economic driven misbehaviors.

Switching mechanism from *T-Opt* to *T-Pes*: The switching mechanism activates only under strict conditions, specifically for detected misbehavior, ensuring it is used when necessary. Typically, *T-Opt* operates as the default mode, focusing on efficiency with low overhead unless misbehavior is detected, which triggers a switch to the resource-intensive *T-Pes*. The executor initiating this switch, typically the reporter of the misbehavior, has their costs covered by the security deposit of the offending party, ensuring honest participants don't bear the cost. Additionally, the system discourages unnecessary *T-Pes* switches. Rational executors won't switch without verified misbehavior, maintaining operational integrity and cost-effectiveness, as they receive no compensation without a genuine breach. This setup keeps executors incentivized to stay in *T-Opt*, optimizing system efficiency and reducing unnecessary overhead.

Allocation proportion of deposits: The allocation of security deposits involves a key trade-off: encouraging misbehavior reporting and preventing dishonest executors from self-reporting for gain. Our protocol first covers the cost executors face when switching from *T-Opt* to *T-Pes*. After this, the remaining deposit is equally divided between reporters and users, fostering a balanced incentive structure that promotes vigilant reporting without giving undue advantage to any party. Furthermore, our protocol accommodates varying security needs and risk tolerances across different applications and users by allowing customization of this allocation proportion. This flexibility ensures the protocol's effective integration into diverse environments, meeting specific security demands.

Step	Function	Gas consumption	Cost in UDS ($)
1	deploy C_p	1114612	5.08
2	*lead()*	797432	3.64
5	*invalid()*	2196769	10.02
6	*follow()*	31198	0.14
8	*leak()*	1264782	5.78
11/16	*execute()*	108542	0.49
12	*deploy()*	2419116	11.04
13	*reveal()*	89727	0.41
14	*missing()*	65766	0.30
15	*fake()*	1279726	5.85

TABLE II: Gas consumption of key functions.

VII. Implementation and evaluation

In this section, we implement and evaluate the proposed T-Watch architecture and protocol. The implementation is designed to run on *rinkeby* [38], the Ethereum official test network where researchers and developers can acquire free *ether* to afford gas consumption of testing protocols and smart contracts. It is widely recognized and supported by recent studies [8], [15], [40] that using test networks, which are designed to closely emulate real-world production environments, is an essential and accepted practice for the preliminary validation of blockchain technologies, providing a safe and cost-effective setting for systematic problem identification and solution refinement. Similar to recent work [11], [14], we first measure gas consumption of key functions in the proposed protocol because the monetary cost of paying gas fee is unavoidable in running services over public blockchains like Ethereum. Then, as introduced in Section III-C, the gas system places greater demand on the scalability of T-Watch, so we vary the size of the executive committee and measure the corresponding cost of running T-Watch through the three execution paths introduced in Section IV-B, respectively. After that, we compare T-Watch with two related works introduced in

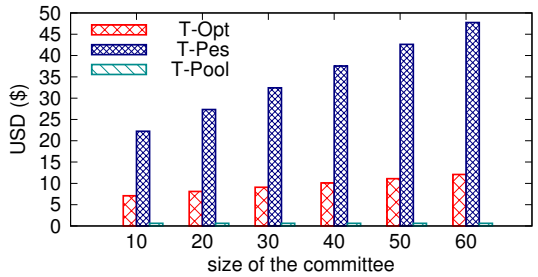


Fig. 6: Cost of *T-Opt*, *T-Pes* and *T-Pool*

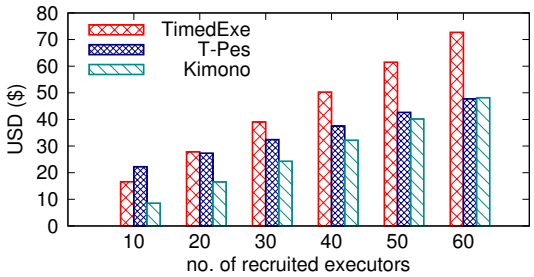


Fig. 7: Comparison against TimedExe and Kimono

Section II, namely TimedExe [26] for timed execution of function invocation transaction, and Kimono [7] for timed release of private data. Finally, T-Watch novelly divides users into two categories, leaders and followers, to further reduce service costs by employing service request pooling. To figure out the effectiveness of this design, we vary the number of followers' requests in a service request pool and measure the corresponding expenses afforded by leaders and followers, respectively. Besides, it is worth noting that there exists an upper bound for the sum of gas usage of all the transactions within a single block and hence, the evaluation of time overhead of executing functions in Ethereum, which is usually on the scale of hundreds of milliseconds, is omitted.

A. Gas consumption of key functions

In TABLE II, we list the key functions in the order that they appear in the protocol shown in Fig. 5. For each key function, we measure its gas consumption, which is then converted into USD for ease of understanding. Concretely, given gas usage c_g, the corresponding cost in USD, namely c_u, could be computed using $c_u = c_g p_g p_e$, where p_g refers to gas price and p_e denotes *ether* price. Since both p_g and p_e fluctuate wildly, we collects their historical values from *Etherscan* [18], and computes their median values over a six-year period from 8/1/2015 to 7/31/2021. According to the results, in the rest of this section, we set p_g and p_e to be 2.29×10^{-8} ether/gas and 199.73 USD/ether, respectively. Besides, we assume a committee formed by 30 executors in this experiment, as well as an execution function *execute()* that will call an existing smart contract and change the value of a single variable there in the rest of this section. As can be seen from the results in TABLE II, there are two types of operations that are more gas-consuming, namely deploying smart contracts and verifying private service keys or random numbers. Specifically, deploying C_p in step 1 costs \$5.08, and deploying C_s through *deploy()* in step 12 spends \$11.04, which demonstrates the need for splitting a single complex contract into the two. We could also see that verifying a leaked or fake sk_i^e via *leak()* or *fake()* spends around \$5.8, and verifying a random number r via *invalid()* spends \$10.02, which indicates the expensiveness of verifying ECDSA keys or ECVRF random numbers on-chain. Among the five remaining functions, *lead()* is the most gas-consuming one because it stores a list of recruited executors in C_b. In contrast, *follow()* is very cheap because it only stores a single address in C_p.

B. Scalability of different execution paths

In this set of experiments, we first match execution paths with functions and then analyze their scalability. First, *T-Opt* requires a leader to deploy C_p (\$5.08) and call *lead()* (\$3.64) during *TT.schedule* to set up a new service request, and later an executor to call *execute()* (\$0.49) during *TT.execute* to complete the service, which results in a total cost of about \$9.21 assuming 30 executors. Besides the \$9.21 above, *T-Pes* additionally requires an executor to invoke *deploy()* (\$11.04) to turn the protocol from *T-Opt* to *T-Pes* and later all the nl executors in $\boldsymbol{E}$ to reveal their private service keys via *reveal()* ($\$0.41nl$), which increases the total cost from \$9.21 to around \$32.55. Different from these two execution paths, *T-Pool* only requires a follower to invoke a single function *follow()* (\$0.14) during *TT.waiting* and hence, its total cost is just \$0.14. Next, to evaluate the scalability of the execution paths, we measure their overall cost by varying the size of the committee from 10 to 60. In Fig. 6, we can see that the results demonstrate our analysis presented in Section IV-B. Concretely, during *TT.schedule*, the cost of uploading addresses of nl executors in *lead()* leads to an increase of about \$0.1 per executor in the overall cost of *T-Opt*. Besides, during *TT.execute*, *T-Pes* needs to afford the additional expenses for uploading nl private service keys via *reveal()*, which further increases the overall cost by around \$0.49 per executor and make *T-Pes* less scalable than *T-Opt*. Finally, we could see that the overall cost of *T-Pool* stabilizes at \$0.14, making *T-Pool* the most scalable execution path among the three.

C. Comparison against TimedExe and Kimono

In Fig. 7, we show the overall cost of *T-Pes*, the least scalable execution path of T-Watch, and that of two related works, TimedExe and Kimono, by varying the number of recruited executors from 10 to 60. We could see that initially, *T-Pes* has the highest cost because the supplemental contract C_s that costs \$11.04 to deploy contains some preventive functions such as *missing()* and *fake()*, as well as some utility functions that help associate C_s with C_p. Then, as the number of executors increases, we could see that the cost of TimedExe and Kimono rises above that of *T-Pes* at the moment of 20 and 60 executors, respectively. The reason is that, *T-Pes* is designed to discourage potential misbehaviors by leveraging the punishment mechanism, while TimedExe and Kimono need to store more data per executor or perform more complicated calculations (e.g., generating shares and recovering secrets) on the blockchain. Finally, by further comparing *T-Opt* and *T-Pool* with the two related works, we

could see that *T-Opt* and *T-Pool* reduces the cost of recruiting 30 executors by over 62% and 97%, respectively, and a service request pool of size 4 reduces the cost by over 90%.

D. Effectiveness of service request pooling

In the last set of experiments, we create service request pools of users of different sizes and evaluate the change in expenses. As shown in Fig. 8, each service request pool consists of a single leader and a varying number of followers, adhering to the protocol described in Section IV-B. This experimental setup validates the practicality of our cost-reduction strategy, as delineated in the *T-Pool* path. Besides, we make the following assumptions: (1) leaders' and followers' requests follow *T-Opt* and *T-Pool*, respectively; (2) the size of the committee is 30; (3) each follower pays \$0.4 worth of *ether* along with *follow()* to offset leader's expenses, as required by the protocol. As can be seen from the results, the cost per follower's request is constant and independent of the service request pool size. After compensation, the adjusted cost of the leader's request decreases as the service request pool size increases and could reach about \$1.49 when there are 19 followers. Finally, we could see that the average cost per request in a service request pool decreases rapidly with the rise in service request pool size. It reduces to about \$3, \$2 and \$1.4 when the service request pool size is 4, 8 and 20, respectively. The results demonstrate the effectiveness of service request pooling and suggest that even a small service request pool can significantly reduce the average cost per request.

E. Off-chain communication cost

As demonstrated in Fig. 4 and 5, off-chain communication in the T-Watch protocol primarily occurs in three steps:

- Step 4, part of the *T-Ops* phase, where the leader U_l sends data to each executor E_i in the committee $\boldsymbol{E}$: $addr(C_p) \parallel C_s \parallel vrs_u \parallel \boldsymbol{o} \parallel E(pk_u, tt_l.payload)$;
- Step 7, during the *T-Pool* phase, where each follower U_f sends their transaction payload to each executor E_i: $addr(C_p) \parallel E(pk_u, tt_f.payload)$;
- Step 9, part of the *T-Ops* phase, where each executor E_i shares their private service key with the rest of the committee $\boldsymbol{E}$: $addr(C_p) \parallel sk_i^e$.

To optimize off-chain communication costs, we have employed the InterPlanetary File System (IPFS) [3] to efficiently transmit data through off-chain channels. Specifically, instead of transmitting complete data such as $addr(C_p) \parallel sk_i^e$, we transmit only the hash values associated with the IPFS stored data, e.g., $hash(addr(C_p) \parallel sk_i^e)$. This approach significantly reduces the volume of off-chain communication while ensuring data integrity and immutability.

Consequently, with k executors and f followers, the off-chain communication cost for *T-Opt* is calculated as $32k + 32k(k-1) = 32k^2$ bytes. For the *T-Pool* phase, specifically in step 7, the off-chain communication cost is $32fk$ bytes.

VIII. Conclusion

This paper proposes T-Watch, the first practical decentralized solution for cost-effectively implementing timed transactions with strong security and scalability guarantees. Our solution employs a novel combination of threshold secret sharing and decentralized smart contracts. To protect the private elements of a scheduled transaction from getting disclosed before the future time-frame, T-Watch maintains shares of the decryption key of the scheduled transaction using a group of executors recruited in a blockchain network before the specified future time-frame and restores the scheduled transaction at a proxy smart contract to trigger the change of blockchain state at the required time-frame. To reduce the cost of smart contracts execution in T-Watch, we carefully design a protocol that offers three execution paths with different characteristics. We rigorously analyze the security of T-Watch and proved that T-Watch is both $\mathcal{A}$-threshold-resistant and $\mathcal{A}$-budget-resistant. Finally, we implement T-Watch over the Ethereum official test network and the results demonstrate that T-Watch is more scalable compared to the state of the art and could reduce the cost of running smart contracts by over 90% through pooling.

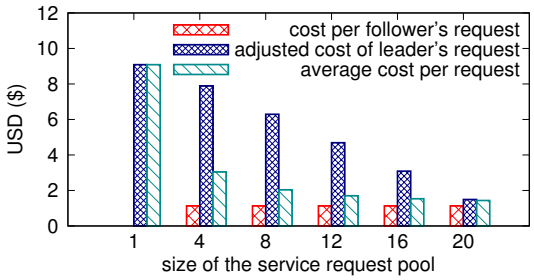


Fig. 8: Effectiveness of service request pooling

Acknowledgement

Chao Li is supported by the Fundamental Research Funds for the Central Universities under Grant No. 2022JBMC007 and the National Natural Science Foundation of China under Grant No. 62202038. Balaji Palanisamy acknowledges the support for this work under Grant #2020071 from the US National Science Foundation (NSF) SaTC program. This material is based upon work supported by the US National Science Foundation under Grant #2020071. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Reference

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

[2] Andreas M Antonopoulos. *Mastering Bitcoin: Programming the open blockchain*. " O'Reilly Media, Inc.", 2017.

[3] Juan Benet. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561*, 2014.

[4] Guido Bertoni et al. Keccak. In *Eurocrypt 2013*, pages 313–314.

[5] Blueorion. https://blueorion.cc/.

[6] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3:37, 2014.

[7] GK Feridun Mert Celebi et al. Kimono:trustless secret sharing using time-locks on ethereum. https://github.com/hillstreetlabs/kimono, 2018.

[8] Fei Chen, Jiahao Wang, Changkun Jiang, Tao Xiang, and Yuanyuan Yang. Blockchain based non-repudiable iot data trading: Simpler, faster, and cheaper. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pages 1958–1967. IEEE, 2022.

[9] Jing Chen et al. Certchain: Public and efficient certificate audit based on blockchain for tls connections. In *IEEE INFOCOM 2018*, pages 2060–2068.

[10] Cryptocurrency statistics. https://bitinfocharts.com/.

[11] Sourav Das, Vinay Joseph Ribeiro, and Abhijeet Anand. Yoda: Enabling computationally intensive contracts on blockchains with byzantine and selfish nodes. *NDSS*, 2019.

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

[13] John R Douceur. The sybil attack. In *International Workshop on Peer-to-Peer Systems*, pages 251–260. Springer, 2002.

[14] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. Perun: Virtual payment hubs over cryptocurrencies. In *2019 IEEE Symposium on Security and Privacy*.

[15] Parinya Ekparinya, Vincent Gramoli, and Guillaume Jourjon. The attack of the clones against proof-of-authority. *NDSS*, 2020.

[16] Keita Emura et al. A timed-release proxy re-encryption scheme. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 94(8):1682–1695, 2011.

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

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

[19] Zhonghui Ge, Yi Zhang, Yu Long, and Dawu Gu. Shaduf: Non-cycle payment channel rebalancing. In *NDSS*, 2022.

[20] Adam Groce and Jonathan Katz. Fair computation with rational players. In *Eurocrypt 2012*, pages 81–98.

[21] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1(1):36–63, 2001.

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

[23] Ahmed Kosba et al. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE symposium on security and privacy*, pages 839–858.

[24] Ranjit Kumaresan et al. How to use bitcoin to incentivize correct computations. In *ACM CCS 2014*, pages 30–41.

[25] Chao Li et al. Nf-crowd: Nearly-free blockchain-based crowdsourcing. In *IEEE SRDS 2020*, pages 41–50.

[26] Chao Li and Balaji Palanisamy. Decentralized privacy-preserving timed execution in blockchain-based smart contract platforms. In *IEEE HiPC 2018*, pages 265–274.

[27] Chao Li and Balaji Palanisamy. Silentdelivery: Practical timed-delivery of private information using smart contracts. *IEEE Transactions on Services Computing*, 2021.

[28] Jia Liu et al. How to build time-lock encryption. *Designs, Codes and Cryptography*, 86(11):2549–2586, 2018.

[29] Mohammad Mahmoody, Tal Moran, and Salil Vadhan. Time-lock puzzles in the random oracle model. In *Annual Cryptology Conference*, pages 39–50. Springer, 2011.

[30] Sai Krishna Deepak Maram et al. Churp: dynamic-committee proactive secret sharing. In *ACM CCS 2019*, pages 2369–2386.

[31] Stephanos Matsumoto and Raphael M Reischuk. Ikp: Turning a pki around with decentralized automated incentives. In *2017 IEEE Symposium on Security and Privacy*, pages 410–426.

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

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

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

[35] Jianting Ning et al. Keeping time-release secrets through smart contracts. *IACR Cryptology ePrint Archive*, 2018.

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

[37] Parity. https://www.parity.io/ethereum/.

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

[39] Ronald L Rivest et al. Time-lock puzzles and timed-release crypto. *Massachusetts Institute of Technology*, 1996.

[40] Muhammad Saad, Songqing Chen, and David Mohaisen. Syncattack: Double-spending in bitcoin without mining power. In *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, pages 1668–1685, 2021.

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

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

[43] Verifiable random functions (vrfs) draft-irtf-cfrg-vrf-12. https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-vrf-12.

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

[45] Jingzhe Wang et al. Attack-resilient blockchain-based decentralized timed data release. In *IFIP WG 11.3 DBSec 2022*, accepted.

[46] Sarisht Wadhwa, Jannis Stoeter, Fan Zhang, and Kartik Nayak. He-htlc: Revisiting incentives in htlc. *NDSS*, 2023.

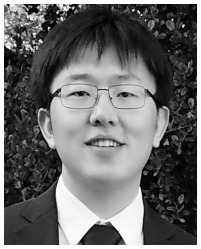
[47] Jingzhe Wang et al. Protecting blockchain-based decentralized timed release of data from malicious adversaries. In *IEEE ICBC 2022*.

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

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

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

[51] Karl Wüst et al. Ace: asynchronous and concurrent execution of complex smart contracts. In *ACM CCS 2020*, pages 587–600.



Chao Li is an Associate Professor in the School of Computer and Information Technology at Beijing Jiaotong University. He received his Ph.D. degree from the School of Computing and Information at University of Pittsburgh and his MSc degree from Imperial College London. His current research interests are focused on Blockchain and Data Privacy.

Balaji Palanisamy is an Associate Professor in the School of computing and information in University of Pittsburgh. He received his M.S and Ph.D. degrees in Computer Science from the college of Computing at Georgia Tech in 2009 and 2013, respectively. His primary research interests lie in scalable and privacy-conscious resource management for large-scale Distributed and Mobile Systems.