SCIENCE CHINA
Information Sciences

March 2020, Vol. 63 130102:1–130102:14
https://doi.org/10.1007/s11432-019-9893-2

• RESEARCH PAPER •
Special Focus on Security and Privacy in Blockchain-based Applications

SecBCS: a secure and privacy-preserving blockchain-based crowdsourcing system

Chao LIN[1,2], Debiao HE[1,2*], Sherali ZEADALLY[3], Neeraj KUMAR[4] & Kim-Kwang Raymond CHOO[5]

[1]*Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University, Wuhan* 430072*, China;*
[2]*State Key Laboratory of Cryptology, P. O. Box* 5159*, Beijing* 100878*, China;*
[3]*College of Communication and Information, University of Kentucky, Lexington* KY 40506*, USA;*
[4]*Department of Computer Science and Engineering, Thapar University, Patiala* 147004*, India;*
[5]*Department of Information Systems and Cyber Security and Department of Electrical and Computer Engineering, University of Texas at San Antonio, San Antonio* TX 78249*, USA*

Received 27 February 2019/Revised 18 April 2019/Accepted 14 May 2019/Published online 11 February 2020

Abstract A robust and scalable crowd management infrastructure is crucial in addressing operational challenges when deploying high-density sensors and actuators in a smart city. While crowdsourcing is widely used in crowd management, conventional solutions, such as Upwork and Amazon Mechanical Turk, generally depend on a trusted third-party platform. There exist several potential security concerns (e.g., sensitive leakage, single point of failure and unfair judgment) in such a centralized paradigm. Hence, a recent trend in crowdsourcing is to leverage blockchain (a decentralized ledger technology) to address some of the existing limitations. A small number of blockchain-based crowdsourcing systems (BCSs) with incentive mechanisms have been proposed in the literature, but they are generally not designed with security in mind. Thus, we study the security and privacy requirements of a secure BCS and propose a concrete solution (i.e., SecBCS) with a prototype implementation based on JUICE.

Keywords crowdsourcing, blockchain, secure, privacy-preserving, smart contract

Citation Lin C, He D B, Zeadally S, et al. SecBCS: a secure and privacy-preserving blockchain-based crowdsourcing system. Sci China Inf Sci, 2020, 63(3): 130102, https://doi.org/10.1007/s11432-019-9893-2

1 Introduction

In smart cities, information and communication technologies are being used to sense, analyze and integrate various data in the daily smooth operation of the city intelligently (e.g., minimal human intervention). Smart city applications range from environmental protection to city services, smart transport, smart grid, and so on [1]. Such an environment typically comprises a large variety of Internet of things (IoT) devices, such as sensors, actuators and other smart devices to sense/collect, transmit and exchange data via the Internet [2]. A typical cosmopolitan city, such as Austin, San Francisco, New York City, and Shanghai, generally has a high density of deployed sensors and actuators. Consequently, such deployment is generally costly and challenging in practice. Hence, we need a robust and scalable crowd management infrastructure to help mitigate some of the existing limitations.

Crowdsourcing, a distributed problem solving model, has attracted considerable interests in crowd management applications. However, popular crowdsourcing systems, such as Upwork[1)] and Amazon

* Corresponding author (email: hedebiao@163.com)
1) https://www.upwork.com/.

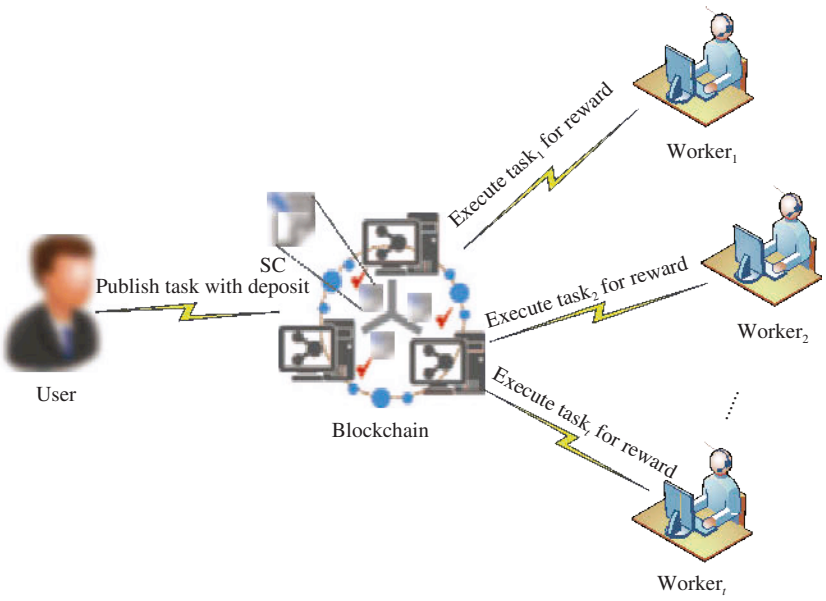

Figure 1 (Color online) Architecture of a typical blockchain-based crowdsourcing system (BCS).

Mechanical Turk[2)], are generally centralized (i.e., depending on a trusted third-party to match the requested/executed tasks pair with fair exchange of rewards). This raises concerns such as sensitive leakage (as sensitive information is stored in the central system), single point of failure (as the centralized system might be temporarily unavailable), and unfair judgment (since there exist no transparent dispute mechanisms). There have been attempts to leverage blockchain (a decentralized ledger technology) to build a blockchain-based crowdsourcing system (BCS) to mitigate these known limitations [3–5]. As shown in Figure 1, a typical BCS mainly consists of three participants (i.e., User, Worker and Blockchain). Here, we take mobile crowd-sensing as an example (i.e., a user can request a bunch of workers to employ their mobile devices to collect information in some data-driven applications). All these requesting/executing task operations (from the user and worker respectively) are achieved by communicating with the blockchain (e.g., triggering the deployed smart contract). The smart contract is a cybernated transaction protocol deployed in the blockchain which performs the terms of a contract (i.e., faithfully handling all computations and message deliveries).

This blockchain-based prototype can achieve both exchange and evaluation fairness without relying on a centralized third-party, but some underlying security and privacy concerns (e.g., identity and data leakage) still exist in its design and implementation. For example, Lu et al. [4] discussed some of these concerns and proposed a private and anonymous BCS. However, their discussions do not include potential security vulnerabilities of a smart contract that can be exploited by an attacker, and their proposal requires a zero-knowledge succinct non-interactive argument of knowledge (zk-SNARK) [6]. The latter has an expensive computational overhead. Hence, we will discuss the various security and privacy requirements that a secure BCS should achieve before proposing a concrete solution without the need for complex and expensive cryptographic primitives such as zk-SNARK.

In Section 2, we will briefly review the related literature.

2 Related work

Crowdsourcing is regarded as a tool for solving distributed problems, which can reduce the cost of production and use human intelligence more effectively. Amazon Mechanical Turk (AMT)[3)] is one of the most popular crowdsourcing systems. However, there are several vulnerabilities (i.e., allowing a

2) https://www.mturk.com/.
3) https://www.mturk.com/mturk/.

false-reporter to obtain ephemeral advantage [7], data leakage for the plaintext form of task answers). To provide privacy protection for AMT, Salehi et al. [8] designed a privacy wrapper (i.e., Dynamo). Although Dynamo can provide anonymity by issuing pseudo IDs (which can only be linked by the issuer), it still inherits all the other deficiencies of AMT.

Li and Cao [9] also adopted a pseudonym method to achieve anonymous crowdsourcing, but their proposal cannot detect malicious workers who pretend pseudo IDs for rewards. Rahaman et al. [10] used group signature to construct an anonymous-yet-accountable crowdsourcing systems. Gisdakis et al. [11] focused on privacy issues during the data crowdsourcing, and introduced more authorities to deal with different functionalities. The distributed authority could reduce the excessive trust, however, the instantiation of these authorities in practice is still intractable.

As discussed in Section 1, there exist some problems (e.g., sensitive leakage, single point of failure, and unfair judgment) in these recently proposed centralized crowdsourcing systems. Hence, several decentralized solutions based on blockchain have been proposed. Li et al. [3] constructed a crowd-shared service using blockchain, however, their proposal provides no incentive or privacy protection. Tanas et al. [12] integrated the blockchain as a payment channel to construct a novel BCS, but their proposal still does not preserve privacy and it cannot provide protection against malicious participants. Several BCSs (e.g., [3,13]) focusing on incentives have been proposed, but these constructions are neither private nor anonymous (i.e., sensitive leakage). The proposal of Lu et al. [4] appears to be the most private and anonymous, but they ignored the vulnerabilities of smart contracts and the required zk-SNARK technology is complex, expensive and has a large proof size.

Next, we will discuss the underpinning security and privacy challenges in BCS.

3 Security and privacy challenges in BCS

We focus mainly on the security and privacy issues (based on the existing studies such as [14–16]) that need to be considered while constructing a secure BCS. A crowdsourcing system, as an information interchange and networking application, should satisfy the underlying security and privacy requirements (i.e., confidentiality, integrity, non-repudiation, availability, access control, and privacy). Although the integration of blockchain can efficiently achieve several of these properties (i.e., integrity, availability, and access control), there are other considerations that need to be taken into account in a blockchain-based crowdsourcing approach.

3.1 Privacy concerns during crowdsourcing task

In a crowdsourcing system, authentication is the basic requirement when requesting (or submitting) a task (or result). This helps to mitigate the risk due to the use of fabricated identities. However, the history of all requests and submissions are published in a decentralized BCS (which was previously guaranteed by a centralized data center). In other words, this would result in leakage of information and compromise user privacy. In some cases (e.g., a user/worker needs to join some frequent traffic surveillance tasks), user location information can be leaked/inferred by analyzing the transactions in the blockchain. A common-prefix-linkable anonymous authentication [4] was proposed to address these privacy concerns. The approach is designed to achieve identifiable identity validation (i.e., only when two authenticated messages are from the same person they can be linked). However, this authentication scheme requires zk-SNARK whose proof generation process is relatively expensive. Although various dedicated optimizations [17,18] for zk-SNARK exist, the cost required to compute the proofs is still not practical for resource-constrained devices.

3.2 On-chained data leakage issues

It is known that the blockchain ledger needs to be replicated in the entire network, which requires the submitted data in the blockchain to be public verifiable for consistency. While this is an attractive property (i.e., transparency) for applications (e.g., supply chain management and data copyright protection),

there are new data privacy challenges in crowdsourcing application, especially when the crowdsourcing data are sensitive. For instance, in the intuitively "safe" image annotation tasks, some responses to special blurred images may be maliciously used to deduce the users' personalities or characteristics (e.g., sexual orientation) thereby comprising one's privacy.

A malicious worker may also obtain rewards without executing the work by simply copying and submitting someone else's uploaded task results as his/her own data because block confirmation is not done in real-time (e.g., generally requires approximately 12 s in Ethereum, after the data is submitted to the network). This will clearly invalidate the incentive mechanisms in the BCS. Hence, data confidentiality should be considered in such a decentralized setting.

Simple standard cryptographic primitives such as encryption may protect the data privacy, but the smart contract will not be able to verify the execution result for enforcing the predefined rewards policy when the data is encrypted. To ensure data confidentiality without affecting availability, an outsource-then-prove methodology [4] was introduced to allow the user to perform his/her verification procedure and then proves the behavior via a valid zk-SNARK proof. However, the associated cost of using zk-SNARK is too high for a resource-constrained device. In addition, the worker's rewards will not be paid until the user submits a zero knowledge proof to trigger the smart contract. Perhaps, one can mitigate the second limitation using a deposit punishment mechanism (e.g., the user pay a fee as a deposit to ensure that he/she will submits the proof in time), but the verification to be executed by the user does not appear to be ideal because this will increase the user's workload especially when dealing with a large number of submissions from the workers.

3.3 Security vulnerabilities in smart contract

Due to the automated execution of contract terms in the smart contract, blockchain 2.0 (supporting smart contract) has been deployed in many applications such as financial applications and asset tracking in IoT [19, 20]. However, security vulnerabilities in such systems can be exploited to facilitate attacks such as the theft of digital assets (e.g., 3.6 million Ethers theft due to the Decentralized Autonomous Organization (DAO) attack) and turning the smart contract into a deadlocked state to prevent the rightful owners from spending or withdrawing their assets. In addition, a smart contract's functionality cannot be tampered with once it is deployed on the blockchain. In other words, vulnerabilities that exist in the contract are difficult to be patched and the corresponding flawed or malicious transactions for triggering the contract will be recorded forever.

Although different approaches for upgradeable smart contracts (e.g., data separation pattern, degatecall-based proxy pattern)[4)] have been proposed, both of which have the potential for flaws with higher complexity and they introduce bugs. This ultimately decreases trust in our smart contracts. Hence, we should strive for simple, unmodifiable, and secure contracts instead of importing codes to defer features and security issues. However, developers (even those are experienced) may casually write security bugs into smart contracts, which means that it is quite difficult to write reliable and safe smart contracts according to the complex semantics of the underlying domain-specific languages and their testability [21].

Thus, we need to develop a secure method (e.g., [22, 23]) to efficiently mitigate this issue. Here, we consider the approach in [23] to be a viable solution for the following reasons. First, the introduced finite-state machine (FSM) based model can efficiently support Ethereum and other systems (e.g., JUICE), which is consistent with our chosen system. Moreover, the user-friendly graphical editor allows one to conveniently design smart contracts such as FSMs, and enables us to design our contract more easily. Finally, the provided easy-to-use tool for translating FSMs into Solidity code has a number of associated plugins for different security requirements. Hence, we choose the approach in [23] for the design of our smart contract. Moreover, we propose using a security testing tool (e.g., Refs. [21, 24], which can empirically evaluate static smart contracts) to further enforce security.

Next, we will review the relevant building blocks in the proposed approach.

4) https://ethereum.stackexchange.com/questions/2404/upgradeable-smart-contracts.

4 System building blocks

4.1 Cryptographic primitives

We will now review the key cryptographic primitives, namely, Group Signatures (GS), Ciphertext-policy Attribute-based Encryption (CPABE) and Advanced Encryption Standard (AES). In both GS and CPABE, some algorithms (e.g., GSetup and ASetup will be introduced subsequently) will need to be executed by a manager, whose responsibility can be taken over by the permissioned nodes in JUICE (as presented in Subsection 3.3).

4.1.1 *Group signatures*

Group signature, first introduced by Chaum and Heyst [25] in 1991, is a special digital signature with anonymity and traceability, and has been considered in many blockchain-based systems to achieve conditional privacy protection. That is, the integration of GS in the transactions of blockchain allows group members to request or execute crowdsourcing tasks on behalf of other group members without exposing themselves. Traceability can be executed by a group manager to identify a misbehaving member. Here, we propose using a group signature scheme of [26] in our design, because its shorter signature length can reduce communication overhead. This scheme consists of five polynomial probability algorithms (i.e., GSetup, Enroll, GSign, GVerify and GTrace), where both GSetup and Enroll are invoked by a group manager to generate a group public key gpk and group members' private keys gsk_i, respectively, GSign is invoked to generate a signature by group members using gsk_i and GVerify can be invoked by anyone to verify a candidate signature using gpk, GTrace is also invoked only by the group manager to trace and repeal the group member identity of a signer respectively. The concrete algorithms are specified as follows.

(1) GSetup: Given a security parameter λ, this algorithm initializes three groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ of prime order q (the length of which is λ bits), a bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ and a SHA hash function $\mathcal{H} : \{0,1\}^* \to \{0,1\}^{256}$. Assuming that the generators of $\mathbb{G}_1$ and $\mathbb{G}_2$ are P_1 and P_2, respectively, it then randomly chooses $d, s, u \in \mathbb{Z}_q$ and computes $D = d \cdot P_1 \in \mathbb{G}_1, S = s \cdot P_2 \in \mathbb{G}_2, U = u \cdot P_1 \in \mathbb{G}_1$. Finally, it returns $\text{PP} = (q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, P_1, P_2, \mathcal{H}(\cdot))$ as the public parameters, $\text{gsk} = (d, s)$ as the group manager's private key, u as the traceability private key and $\text{gpk} = (D, S, U)$ as the group public key.

(2) Enroll: Given public parameters PP and the group manager's private key $\text{gsk} = (d, s)$, this algorithm randomly chooses $x_i \in \mathbb{Z}_q$ and computes $Z_i = (d - x_i)(sx_i)^{-1} \cdot P_1$, as well as $\text{tag}_i = \mathcal{H}(x_i \cdot Z_i)$. It returns $\text{gsk}_i = (x_i, Z_i)$ as a group member's private key, and tag_i as the label. We note that the group manager stores the group member's identity ID_i with tag_i into a list List.

(3) GSign: Given public parameters PP, the group public key $\text{gpk} = (D, S, U)$, a group member's private key $\text{gsk}_i = (x_i, Z_i)$ and a message M, this algorithm randomly chooses $k \in \mathbb{Z}_q$ and computes $C_1 = k \cdot P_1 \in \mathbb{G}_1, C_2 = x_i \cdot Z_i + k \cdot U \in \mathbb{G}_1, Q = e(U, S)^k \in \mathbb{G}_T$. It also computes $\text{digest} = \mathcal{H}(M)$ and $c = \mathcal{H}(C_1, C_2, Q_{\text{digest}})$, as well as $w = kc + x_i \in \mathbb{Z}_q$. It returns the group signature $\sigma = (C_1, C_2, c, w)$.

(4) GVerify: Given public parameters PP, the group public key $\text{gpk} = (D, S, U)$ and a candidate message/signature pair $(M, \sigma = (C_1, C_2, c, w))$, this algorithm computes

$$\widetilde{Q} = \frac{e(C_2, S) \cdot e(w \cdot P_1, P_2)}{e(c \cdot C_1 + D, P_2)},$$

$\text{digest} = \mathcal{H}(M)$. If $c = \mathcal{H}(C_1, C_2, \widetilde{Q}, \text{digest})$, then it returns 1 to accept σ.

(5) GTrace: Given public parameters PP, the traceability private key ok, the list of group members List and the candidate signature $\sigma = (C_1, C_2, c, w)$, this algorithm computes $\text{tag}_i = \mathcal{H}(x_i \cdot Z_i) = \mathcal{H}(C_2 - u \cdot C_1)$. Then it retrieves the List to determine the identity ID_i of the signature σ.

4.1.2 *Ciphertext-policy attribute-based encryption*

In our design, we adopt CPABE [27] to protect the confidentiality of a crowdsourcing task. That is, only someone whose attributes satisfy the predefined access polices can correctly execution the decryption.

It comprises four polynomial probability algorithms (i.e., ASetup, ABEnc, AKeyGen, ABDec), where the ASetup algorithm is invoked by the manager to generate a master private key msk and public parameters apk; anyone who has obtained apk can encrypt a message with a preset access polices by invoking ABEnc, and the receiver who receive the ciphertext can execute the decryption via ABDec after requesting the decryption key (related to its attributes) from the manager (who owns the msk invoking AKeyGen to reply). In addition to the above expressive functionality, the adopted scheme also achieves efficiency and provable security. We describe the concrete algorithms next.

(1) ASetup: Given a security parameter λ and the amount of attributes U in this system, this algorithm initializes a group $\mathbb{G}$ of prime order q (the length of which is λ). Assuming that the group G is generated by P, it randomly chooses U group elements $Q_1, \ldots, Q_U \in \mathbb{G}$. It also chooses two randomnesses $\alpha, a \in \mathbb{Z}_q$ and computes $\mathcal{A} = e(P,P)^{\alpha}, A = a \cdot P$. It returns apk $= (P, \mathcal{A}, A, Q_1, \ldots, Q_U)$ as the public key and msk $= \alpha \cdot P$ as the master private key.

(2) ABEnc: Given the public key apk, a message M, and an LSSS access structure $(\mathbb{M}, \rho)$, where ρ is a function associating rows of $\mathbb{M}$ to attributes and $\mathbb{M}$ is an $l \times n$ matrix, this algorithm randomly chooses a vector $\boldsymbol{v} = (s, y_2, \ldots, y_n) \in \mathbb{Z}_q^n$ for sharing the encryption exponent s. Then it computes $\lambda_i = \boldsymbol{v} \cdot \mathbb{M}_i$ for $i = 1, \ldots, l$, where $\mathbb{M}_i$ is the vector corresponding to the ith row of $\mathbb{M}$. Next, it randomly chooses $r_1, \ldots, r_l \in \mathbb{Z}_q$ and computes $C = M\mathcal{A}^s, C' = sP, C_i = \lambda_i A + (-r_i)Q_{\rho(i)}, D_i = r_i P$, where $i = 1, \ldots, l$. It returns CT $= (C, C', (C_i, D_i)_{i=1}^{l}, \mathbb{M}, \rho)$ as the ciphertext.

(3) AKeyGen: Given the master private key msk and an attribute set S, this algorithm randomly chooses $t \in \mathbb{Z}_q$ and computes $K = \alpha P + tA, L = tP, K_x = tQ_x$ for $\forall x \in S$. It returns ask $= (K, L, K_x)$ $(\forall x \in S)$ as the attribute private key.

(4) ABDec: Given a ciphertext CT under access structure $(\mathbb{M}, \rho)$ and a private key ask for attribute set S. Assuming that S matches the access structure and we denote $I \subset \{1, 2, \ldots, l\}$ as $I = \{i : \rho(i) \in S\}$. We denote $\{w_i \in \mathbb{Z}_q\}_{i \in I}$ as a set of integers, where $\Sigma_{i \in I} w_i \lambda_i = s$ if each $\{\lambda_i\}$ is valuable share of any secret s based on $\mathbb{M}$. This algorithm computes

$$\frac{e(C', K)}{\Pi_{i \in I}(e(C_i, L)e(D_i, K_{\rho(i)}))^{w_i}} = \frac{e(P,P)^{\alpha s} e(P,P)^{ast}}{\Pi_{i \in I} e(P,P)^{ta\lambda_i w_i}} = e(P,P)^{\alpha s}$$

by C to get the message M. It returns the message M.

4.1.3 *Other encryption solutions*

We leverage AES [28] (a typical symmetric encryption scheme) to protect the confidentiality of crowdsourcing task results, i.e., all crowdsourcing task results are encrypted under AES, such that only the requestor who owns the secret key can decrypt and obtain the task results. We also recommend that any secure asymmetric encryption technology (mainly comprising a public/private key-pair generation algorithm KG, an encryption algorithm Enc, and a decryption algorithm Dec) can be used to ensure privacy of intermediate data (e.g., verify code or task execution results) during the crowdsourcing process.

4.2 Trusted execution environment (TEE)

The trusted execution environment [29] (TEE) is a secure and isolated execution environment provided by recent commodity CPUs, which can provide the confidentiality and integrity of loaded code and data. Here, isolation means that TEE is run in parallel with the operating system (OS). The security of applications in TEE (e.g., code or data) is guaranteed by reducing the trusted computing base to only the CPU, and secure memory and cryptographic primitives provide further protection against other internal applications, as well as privileged software (e.g., OS or supervisory program).

According to the above features of TEE, we use software guard extensions (SGX)[5)] (one of the TEE paradigms) in our design. Its trusted and isolated environment (enclave) can be used for the remote

5) Intel Corp. Intel software guard extensions Software Development Kit (SDK). https://software.intel.com/en-us/sgx-sdk.

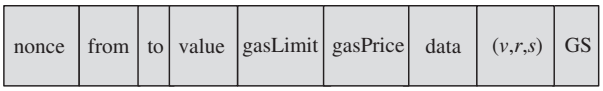

Figure 2 The structure of a request transaction.

deployment of crowdsourcing verification procedures. This can guarantee its security (including confidentiality and integrity) because there is protection from other malicious procedures when running inside the enclave. In addition, SGX will generate a remote attestation after executing the procedures, which allows a remote user (i.e., smart contract in our design) to verify its correct execution in the enclave. It is worth noting that SGX can also generate a secure channel for others to communicate with it thereby allowing us to build a secure communication between the user (or worker) and TEE in our work.

4.3 JUICE

JUICE is an open service platform, which can support Solidity such as Ethereum and build a friendly graphical interface using Java and JavaScript for the user. In addition, it has been used in privacy-preserving applications such as [30] due to its diverse cryptographic application programming interface (API) calls (e.g., homomorphism encryption, group signature, and zero-knowledge proof).

The transaction format of JUICE is modified from Ethereum and hence, it is similar to that of Ethereum. The most complex difference is the adopted cryptographic API calls, and the transaction in our proposal mainly includes attribute-based encryption and group signature. As shown in Figure 2, the fields (i.e., "nonce", "from", "to", "value", "gasLimit", "gasPrice" and "(v, r, s)") in our proposal are identical to those in Ethereum but the extended "GS" represents a group signature for anonymous authentication and traceability if such features are needed. In addition, "data" here comprises "Hash(verifyCode)" and "ABEnc(task||reward)" (when the transaction is a request of publishing a crowdsourcing task) but consists of "VS" and "Enc(key, ret)" (when the transaction is a submitted result from the worker), where "Hash(verifyCode)" is the hash digest of validation procedures, "ABEnc(task||reward)" is the attributed-based encryption of a crowdsourcing task and its reward, "VS" is a verification signature generated by the hardware enclave, and "Enc(key, ret)" is the ciphertext via AES encryption of the execution results. Note that the complex cryptographic primitives (including CPABE and AES) could be executed off-chain and the integrated group signature in the transaction could be realized by modifying the basic layer of the cryptographic algorithm libraries in Ethereum (i.e., JUICE).

If a user (or a worker) wishes to publish a crowdsourcing task (or submits the execution results), the user needs to prepare and broadcast the above transaction. All transactions will be chronologically chained in the blockchain network by some permissioned nodes, according to PBFT [31] or RAFT [32] (which are the consensus mechanisms used in JUICE).

4.4 Secure smart contract design

The logic unit is based on only one single smart contract (called smart contract on CST), which mainly consists of uploadTask, uploadPK, getTask, executeTask and getResult algorithms. A user can invoke uploadTask to upload a crowdsourcing task with an advance and some workers can get a task via getTask. Then, the worker can submit the correct execution result (including TEE's validation signature and the task result) via executeTask to receive the reward and the user can retrieve the result via getResult. It is worth noting that the uploadPK algorithm is performed by a user to upload TEE's chosen public key for the current task (which is used for the security of subsequently transmitted data to TEE) and the encrypted symmetric key (which ensures only the user can decrypt the execution results). We describe the details of smart contract on CST later.

Prior work for solving the potential vulnerabilities when deploying smart contracts in practice mainly provides tools to verify correctness or identify common vulnerabilities. Here, we suggest using FSolidM [23] (a finite state machine based approach) with an empirical evaluation (e.g., [21, 24]) to design our secure smart contract above. This is an alternative technology (i.e., other techniques could also be integrated into

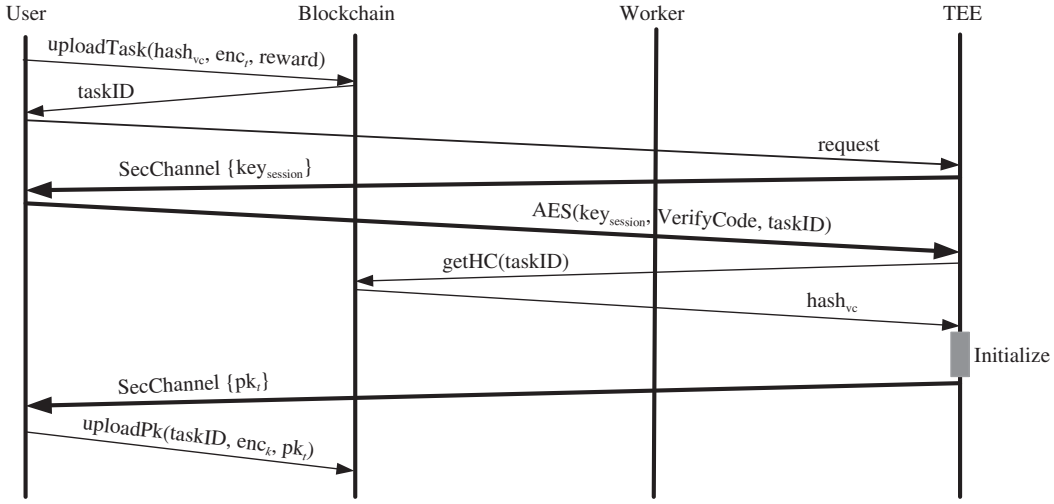

Figure 3 Deployment of publishing a crowdsourcing task.

our construction), which performs rigorous semantic checks by designing smart contracts as finite state machines (FSMs). Hence, this efficiently supports smart contracts in Ethereum and JUICE. FSolidM also provides a user-friendly graphical editor that facilitates the design of smart contracts as FSMs, and a tool for translating FSMs into Solidity code.

5 Proposed architecture

Next, we describe SecBCS (mainly comprises publishing crowdsourcing tasks and executing crowdsourcing tasks) based on the above system building blocks. In addition to the participants (i.e., User, Worker and Blockchain) in the conventional BCS architecture, our design introduces another TEE for secure execution of the crowdsourcing verification procedures. We note that the parameters involved in both GS and CPABE will be initialized by permissioned nodes in JUICE. That is, permissioned nodes invoke GSetup, Enroll and ASetup algorithms to initialize system public parameters and also generate corresponding public/private keys. All the transactions involved in the following phases will be generated via GS by default, to address identity privacy concerns.

5.1 Publishing crowdsourcing tasks

When a user (e.g., Alice) wants to publish a crowdsourcing task, she performs two sub-tasks (i.e., submits her required task data to the smart contract and transmits the corresponding validation procedure to TEE). At the conclusion of this process, an asymmetric key pair will be generated which can be used to build secure communications in the following interactions without the need to establish a secure communication channel. The details are depicted in Figure 3 and are described as follows.

• Alice prepares the predicate, encrypts her crowdsourcing task via ABEnc algorithm (denoted by enc_t), and computes the hash of its corresponding validation procedure (denoted by hash_{vc}), prior to invoking the uploadTask algorithm to upload the required task data (which also includes a reward for this task, i.e., reward). Note that this reward is also as the incentive for workers to execute the task. She will then obtain a taskID for subsequently updating or querying of the task states.

• After the data uploading, Alice uses a secure communication channel (built by the remote attestation from TEE and protected by AES using a $\text{key}_{\text{session}}$) to send the corresponding validation procedure (denoted by VerifyCode) and identity of task (denoted by taskID) to TEE. TEE decrypts the encrypted data and obtains VerifyCode and taskID, and receives hash_{vc} by invoking the getHC algorithm to determine the integrity of VerifyCode.

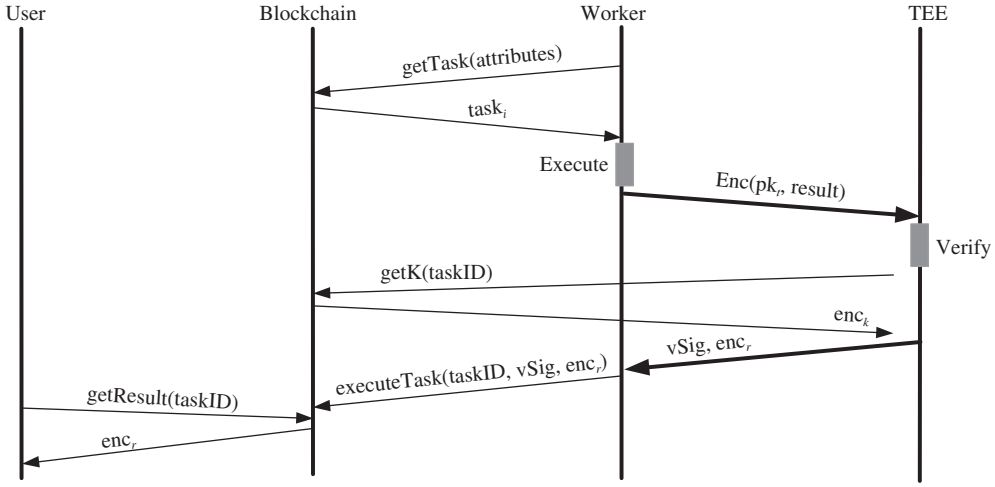

Figure 4 Procedure for executing a crowdsourcing task.

• If the VerifyCode is valid, then TEE initializes an asymmetric encryption public/private key pair $(\mathrm{pk}_t, \mathrm{sk}_t)$ (i.e., by invoking the KG algorithm) and sends pk_t to Alice; otherwise, TEE terminates the process. After receiving the pk_t, Alice uses it to encrypt her secret key (i.e., by invoking $\mathsf{ENC}(\mathrm{pk}_t, \mathrm{key})$ to obtain enc_k, where key is chosen by herself for encrypting the final execution results) followed by uploading enc_k and pk_t into the smart contract by invoking the uploadPK algorithm.

5.2 Executing crowdsourcing tasks

Any worker (e.g., Bob) can obtain and execute the crowdsourcing tasks from the smart contract in blockchain. Figure 4 shows the procedure.

• Bob invokes getTask using his attributes to obtain the crowdsourcing tasks. The smart contract returns the matched and non-finished tasks to Bob (where Bob invokes ABDec to obtain the plaintext of tasks). Then, Bob executes the tasks.

• When Bob finishes a task, he verifies the correctness of the execution result (denoted by result) with the help of TEE. He first uses pk_t to encrypt the result via Enc and sends the encrypted data to TEE. Then, TEE executes the decryption using Dec and his sk_t and verifies the correctness of result according to the pre-deployed VerifyCode.

• After the validation, TEE queries the encrypted key enc_k from the smart contract via getK. Then, it encrypts the result via AES using the key obtained by decrypting enc_k via DEC. Here, we denote the encrypted result as enc_r. TEE replies vSig and enc_r to Bob, where vSig is an attestation generated by TEE containing the claimed validation result (i.e., whether the execution result is true or not).

• Finally, Bob submits taskID, vSig and enc_r to the smart contract via executeTask and obtains the reward if the execution result is valid (which means that the vSig is valid and its claimed validation result is true). Correspondingly, the user (e.g., Alice) can retrieve the enc_r from the smart contract and decrypt it to obtain the final result using key.

6 Security analysis

In this section, we first explain how the proposed solution can satisfy the aforementioned security and privacy requirements.

Privacy protection with accountability. In our design, each task request or execution uses a group signature for authentication. Due to the anonymity of the group signature, any group member's true identity will not be revealed. Hence, their identity privacy is ensured. The traceability property of the used group signature scheme guarantees that only the group manager can trace a suspicious or malicious transaction to the concrete group member via GTrace. If one (who is not the group manager)

Table 1 Comparison between our proposal with other crowdsourcing systems

Feature	SecBCS (proposed)	AMT[6)]	Dynamo [8]	CrowdBC [3]	ZebraLancer [4]
Data confidentiality	√	×	×	×	√
User anonymity	√	×	√	×	√
Traceability	√	√	√	√	×
Secure deployment	√	–	–	×	×
Fair judgment	√	×	×	√	√
Reliability	√	×	×	√	√

would like to trace a group signature, he/she needs to break the security of the underpinning Elgamal encryption. Hence, we can efficiently protect the identity privacy of participants without compromising accountability.

Data security with verifiability. The adopted CPABE in our architecture can provide the confidentiality of the published task in the smart contract. Only when someone's attributes satisfy the predefined access control policy can he/she execute the decryption and obtain the task's information. This allows us to achieve data protection with the functionality of pre-specifying the range of a receiver (i.e., someone whose attributes satisfy the predefined access polices), which can be put into practice in the implementation of BCS. In addition, a simple symmetric encryption scheme (i.e., AES) can be used to protect the task's execution results.

We can further ensure the privacy of these results, which is due to the integration of TEE. Anyone seeking to leak the execution results needs to corrupt SGX, which will require the successful execution of side-channel attacks to steal information from the enclave. Hence, we can effectively resist these attacks by simply limiting the execution times. Note that our proposal is not limited to Intel hardware. Any other trusted hardware can be used in our architecture.

Secure deployment of a smart contract. This requirement is intuitively satisfied due to the FSolidM [23] method used for designing a smart contract. FSolidM supported plugins can implement security features to prevent common vulnerabilities (i.e., reentrancy and unpredictable state) and implement common design patterns for deploying correct contracts with complex functionality. Here, we also emphasize that human factors should never be ignored even using such a secure tool for the deployment of smart contracts.

Next, we compare our SecBCS with some existing crowdsourcing systems to analyze the security performance of our proposal. We denote √ as a supported feature, × as an unsupported feature. The – means that the scheme does not involve this feature. We note that "Data confidentiality" requires that only the requesters can access the submitted data, "Secure deployment" represents the secure deployment of smart contracts, "Reliability" refers to the resistance to a single point of failure (considering the running of crowdsourcing systems). As shown in Table 1, in addition to the data confidentiality, user anonymity, fairness of exchange and reliability, our proposal also achieves traceability and secure deployment. These features are essential for a BCS before its actual deployment.

7 Implementation and performance evaluation

To demonstrate the feasibility of SecBCS, we implemented it on JUICE (client version). Considering that JUICE does not provide the API of ABE at the time of this study, we use the off-chained execution of the cryptographic primitives (i.e., GS, CPABE) involved. In addition, both Intel SGX and FSolidM tools are mature enough to be directly used in practice. Crowdsourcing is a very broad term and each crowdsourcing task (e.g., collecting information, processing data) has its own task description with corresponding verification procedures. An extensive design of these procedures is beyond the scope of this work. Hence, we focus mainly on the deployment of the JUICE test chain[7)] and the logic design of

6) https://www.mturk.com/

7) The JUICE platform is a one-stop service platform of blockchain, which could be accessed at https://open.juzix.net/download/.

Table 2 Testing platform information

Operating system	Ubuntu 16.04
CPU	Intel (R) core (TM) i7-6700 CPU @3.40 GHz
Memory	3 GB RAM
Configuration	Naginx-1.11.3; truffle-4.1.13; JUICE-client

Table 3 Smart contract gas cost (gas price = 2 Gwei, 1 ether = 122.22 USD)

Operation	Gas used	Actual cost (ether)	USD
publishing	1119349	0.002238698	0.2736
uploadTask	275879	0.000551758	0.0674
uploadPK	27041	0.000054082	0.0066
getTask	23277	0.000046554	0.0057
getHC	243877	0.000048774	0.0060
getK	24343	0.000048686	0.0060
getResult	24409	0.000048818	0.0060
executeTask	42469	0.000084938	0.0104

a smart contract.

We constructed the test chain using our personal computer (as shown in Table 2), where nginx-1.11.3 is a high performance hyper text transfer protocol (HTTP) and reverse proxy server, truffle-4.1.13 is a development framework for blockchains using the Ethereum Virtual Machine (EVM), and JUICE-client is a client version of JUICE platform. After the deployment of the test chain, we implemented the logic units design of Smart Contract on CST (shown in Algorithms 1 and 2). We used Solidity to simplify the implementation. However, in a real-world deployment, we recommend using FSolidM for a more secure design of a smart contract. For convenient verification of the signature from the TEE, we used the elliptic curve digital signature algorithm (ECDSA) scheme such that the existing cryptographic function ecrecover[8] in Solidity can be directly invoked in the smart contract. After the deployment of our smart contract[9] in the test chain, we used Web3j[10] (a lightweight Java and Android library for working with smart contracts and embedding in client nodes of Ethereum) to evaluate the functionality of updateTask, getHC, uploadPK, getTask, executeTask, getK, and getResult.

We first computed the gas cost of these operations, and it appears that the costs for multiple executions are almost unchanged (as shown in Table 3). It is worth noting that the publishing operation of CST was executed only once and the cost was approximately USD 0.2736, but all the other operations would be performed in each overall process of the crowdsourcing task. The balance would be checked to determine if it is sufficient for these executions, which is a lower cost to accept in comparison to the cost of a crowdsourcing task. In addition to the gas cost of these operations, we studied the time cost under different number of concurrent transactions. Hence, we run the test by changing the number of concurrent transactions ranging from 50 to 500 and recorded the corresponding average time cost results. As shown in Figure 5, the graphs associated with uploadTask, uploadPK and executeTask fluctuate, but their ranges tend to one particular value (i.e., 7, 7, and 12 s, respectively). In addition, the time costs of getHC, getK and getResult change slowly and are not higher than 2 s. This is consistent with the consensus difficulty adjustment of Ethereum. The average time cost here not only refers to the time it takes to confirm a block, but also includes the transaction generation time, sealing block's time and the cryptographic operations time (e.g., invoking ecrecover in the executeTask function which introduces another time cost). This is why the change of some graphs shows a "wave" variation. Such a delay can be reduced by designing a specialized platform for SecBCS, which is one of our future work.

8) https://solidity.readthedocs.io/en/v0.5.4/.

9) The link of our smart contract design is https://github.com/colyn91/Smart-Contract-on-CST/tree/master/truffle_crowdsourcing.

10) https://docs.web3j.io/.

Algorithm 1 Part 1 — smart contract on CST

```
Require: Function name, invoked parameters;
Ensure: Setting up functions:
structure CSTask
% The structure of a crowdsourcing task.
  taskID; % Identity of a task.
  predicate; % Predicate in the attribute-based encryption.
  hash_vc; % Hash of the crowdsourcing verification procedure.
  enc_t; % Ciphertext of task under attribute-based encryption.
  reward; % Reward of executing the task.
  pk_t; % Task of unique public key generated by TEE.
  enc_k; % Ciphertext of the user's symmetric key encrypted by pk_t.
  enc_r; % Ciphertext of the task result encrypted by user's symmetric key.
  state; % Task state represents whether it has been executed or not.
function CST()
% Constructor, automatically invoked when this contract is deployed.
  CSTask [] public task;
  CSTask tmpTask;
function payable uploadTask(hashVerifyCode, predicate, encTask, taskPrice)
  % A conditional function invoked by the user to upload a crowdsourcing task by paying the claimed reward.
  (msg.value == taskPrice); % Consistency requirement of payment.
  tmpTask.taskID = random();
  tmpTask.predicate = predicate;
  tmpTask.hash_vc = hashVerifyCode;
  tmpTask.enc_t = encTask;
  tmpTask.state = false;
  task.push(tmpTask);
  return tmpTask.taskID;
function public uploadPK(taskID, publicKey, encKey)
% Invoked by a user to upload the TEE's chosen public key.
  r = Find(taskID);
  task[i].pk_t = publicKey;
  task[i].enc_k = encKey;
function public getTask(attributes)
% Invoked by the worker to obtain a crowdsourcing task.
  CSTask[] public tmp;
  i = 0;
  while i < task.length do
    if task[i].state == false and attributes meet task[i].predicate then
      tmp.push(task[i]);
    end if
    i++;
  end while
  return tmp;
function public executeTask(taskID, vSig, encResult)
% Invoked by the worker to upload the result and obtain the reward.
  r = Find(taskID);
  if vSig is valid and the claimed result is true then
    task[r].state = true;
    task[r].encr = encResult;
    msg.sender.transfer(task[r].taskPrice);
  else
    task[r].state = false;
  end if
```

8 Conclusion and future work

The blockchain-based crowdsourcing system can potentially overcome the challenges in solving complex tasks in the deployment of high-density sensors and actuators in a smart city. Ensuring the security and privacy of on-chained data (i.e., crowdsourcing tasks and their execution results) in BCS is one of several

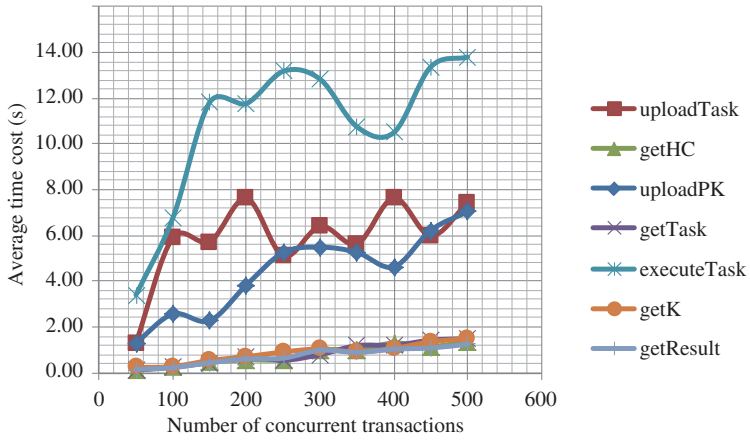

Figure 5 (Color online) Variation of the average time cost with the number of concurrent transactions.

Algorithm 2 Part 2 — smart contract on CST

function public getHC(taskID)
% Invoked by the user to gain hash of the crowdsourcing verification procedure.
r = Find(taskID);
return task[r].hash$_{vc}$;
function public getResult(taskID)
% Invoked by the user to obtain the task results.
r = Find(taskID);
return task[r].enc$_r$;
function public getK(taskID)
% Invoked by the user to obtain the ciphertext of the user's symmetric key.
r = Find(taskID);
return task[r].enc$_k$;

key requirements and challenges. Hence, we proposed a novel architecture called SecBCS (a secure and privacy-preserving BCS with incentive mechanisms). We also presented a prototype of SecBCS on JUICE to evaluate its utility in practice.

In the future, we will extend SecBCS to include other approaches and explore the applications of SecBCS. For example, the architecture of SecBCS with incentive mechanisms and the security and privacy properties can be applied to other consumer applications, such as decentralized storage services, and e-commerce. There are also many complex crowdsourcing scenarios (e.g., crowdsourcing for bioinformatics or geospatial data), which require the design of a highly efficient evaluation mechanism for different specific settings in SecBCS. In addition, our proposal is based on JUICE (maintained by some permissioned nodes), which requires trusted participants. Such a requirement may not be practical in some sensitive applications or adversarial environments. Hence, one future research direction of this work is the design and development of a decentralization approach.

Acknowledgements This work was supported in part by National Key Research and Development Program of China (Grant No. 2018YFC1604004), National Natural Science Foundation of China (Grant Nos. 61572379, 61772377, 61841701), and Natural Science Foundation of Hubei Province of China (Grant Nos. 2017CFA007, 2015CFA068). The last author is supported by Cloud Technology Endowed Professorship. We thank the anonymous reviewers for their valuable comments and feedback which helped us to improve the content and presentation of this paper.

References

1 Su K, Jie L, Fu H. Smart city and the applications. In: Proceedings of International Conference on Electronics, Ningbo, 2011. 1028–1031
2 Zanella A, Bui N, Castellani A, et al. Internet of things for smart cities. IEEE Internet Things J, 2014, 1: 22–32
3 Li M, Weng J, Yang A, et al. CrowdBC: a blockchain-based decentralized framework for crowdsourcing. IEEE Trans Parallel Distrib Syst, 2019, 30: 1251–1266

4 Lu Y, Tang Q, Wang G. Zebralancer: private and anonymous crowdsourcing system atop open blockchain. In: Proceedings of the 38th IEEE International Conference on Distributed Computing Systems, Vienna, 2018. 853–865
5 Buccafurri F, Lax G, Nicolazzo S, et al. Tweetchain: an alternative to blockchain for crowd-based applications. In: Proceedings of the 17th International Conference on Web Engineering, Rome, 2017. 386–393
6 Ben-Sasson E, Chiesa A, Genkin D, et al. Snarks for C: verifying program executions succinctly and in zero knowledge. In: Proceedings of Advances in Cryptology-CRYPTO, California, 2013. 90–108
7 McInnis B, Cosley D, Nam C, et al. Taking a HIT: designing around rejection, mistrust, risk, and workers' experiences in amazon mechanical turk. In: Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, California, 2016. 2271–2282
8 Salehi N, Irani L C, Bernstein M S, et al. We are dynamo: overcoming stalling and friction in collective action for crowd workers. In: Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, Seoul, 2015. 1621–1630
9 Li Q, Cao G. Providing efficient privacy-aware incentives for mobile sensing. In: Proceedings of IEEE 34th International Conference on Distributed Computing Systems, Madrid, 2014. 208–217
10 Rahaman S, Cheng L, Yao D D, et al. Provably secure anonymous-yet-accountable crowdsensing with scalable sublinear revocation. Proc Privacy Enhancing Technol, 2017, 2017: 384–403
11 Gisdakis S, Giannetsos T, Papadimitratos P. Security, privacy, and incentive provision for mobile crowd sensing systems. IEEE Internet Things J, 2016, 3: 839–853
12 Tanas C, Delgado-Segura S, Herrera-Joancomartí J. An integrated reward and reputation mechanism for MCS preserving users' privacy. In: Proceedings of the 10th International Workshop and the 4th International Workshop, Vienna, 2015. 83–99
13 Muehlemann A. Sentiment protocol: a decentralized protocol leveraging crowd sourced wisdom. 2017. ArXiv: 1710.11597
14 Lin C, He D B, Huang X, et al. A new transitively closed undirected graph authentication scheme for blockchain-based identity management systems. IEEE Access, 2018, 6: 28203–28212
15 Feng Q, He D B, Zeadally S, et al. A survey on privacy protection in blockchain system. J Netw Comput Appl, 2019, 126: 45–58
16 Lin C, He D B, Huang X Y, et al. Blockchain-based system for secure outsourcing of bilinear pairings. Inf Sci, 2018. doi: 10.1016/j.ins.2018.12.043
17 Veeningen M. Pinocchio-based adaptive zk-SNARKs and secure/correct adaptive function evaluation. In: Proceedings of the 9th International Conference on Cryptology, Senegal, 2017. 21–39
18 Kosba A, Miller A, Shi E, et al. Hawk: the blockchain model of cryptography and privacy-preserving smart contracts. In: Proceedings of the 37th IEEE Symposium on Security and Privacy, California, 2016. 839–858
19 Christidis K, Devetsikiotis M. Blockchains and smart contracts for the Internet of things. IEEE Access, 2016, 4: 2292–2303
20 Zhang Y, Wen J. The IoT electric business model: using blockchain technology for the Internet of things. Peer-to-Peer Netw Appl, 2017, 10: 983–994
21 Parizi R M, Dehghantanha A, Choo K K R, et al. Empirical vulnerability analysis of automated smart contracts security testing on blockchains. In: Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering, Markham, 2018. 103–113
22 Luu L, Chu D H, Olickel H, et al. Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, New York, 2016. 254–269
23 Mavridou A, Laszka A. Designing secure ethereum smart contracts: a finite state machine based approach. 2017. ArXiv: 1711.09327
24 Parizi R M, Singh A, Dehghantanha A. Smart contract programming languages on blockchains: an empirical evaluation of usability and security. In: Proceedings of the 1st International Conference on Blockchain Blockchain, Washington, 2018. 75–91
25 Chaum D, van Heyst E. Group signatures. In: Proceedings of Advances in Cryptology-EUROCRYPT, Brighton, 1991. 257–265
26 Ho T H, Yen L H, Tseng C C. Simple-yet-efficient construction and revocation of group signatures. Int J Found Comput Sci, 2015, 26: 611–624
27 Waters B. Ciphertext-policy attribute-based encryption: an expressive, efficient, and provably secure realization. In: Proceedings of 14th International Conference on Practice and Theory in Public Key Cryptography, Taormina, 2011. 53–70
28 Daemen J, Rijmen V. The Design of Rijndael: AES - The Advanced Encryption Standard. Berlin: Springer, 2002
29 Poulpita. Trusted execution environment, millions of users have one, do you have yours? 2017. https://poulpita.com/2014/02/18/trusted-execution-environment-do-you-have-yours/
30 Lin C, He D, Huang X, et al. BSeIn: a blockchain-based secure mutual authentication with fine-grained access control system for industry 4.0. J Netw Comput Appl, 2018, 116: 42–52
31 Castro M, Liskov B. Practical byzantine fault tolerance. In: Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation, Louisiana, 1999. 173–186
32 Ongaro D, Ousterhout J. In search of an understandable consensus algorithm. In: Proceedings of 2014 USENIX Annual Technical Conference, Philadelphia, Pennsylvania, 2014. 305–319