zkPoD: A Practical Decentralized System for Data Exchange

YUGUANG HU, X.M. SUN, YU GUO, ZHIPENG SUN, YIWEN LU, SHAOFAN WANG, CUN YE, YUN LI, and CHAO ZHANG

Online commerce traditionally needs trusted third parties (TTP) to ensure the fairness of the trading, *i.e.,* each party gets the other's item, or neither party does. However, TTPs tend to be a centralized entity to guarantee the trust, which inevitably increases risks of both privacy leaking and functional failure. Reducing the dependence on TTPs while keeping fairness is a classic open problem for decades. The strong fairness, ensuring neither party of the buyer and seller cannot take any advantage over the other, will help to build trustless trading system that can be highly effective. Participants are able to exchange digitalized commodities (or data) equally and freely without concern about mistrust.

In this paper, we present cryptographic protocols, proof-of-delivery (PoD), to solve a fundamental problem of fair data exchange. The protocols ensure that a chunk of data should be delivered faithfully by using a blockchain, which is not only a trustless third party doing public verification, but also provides cryptocurrencies to support (strong) fair data exchange, or data trading. We present three variants, PoD-AS, PoD-AS* and PoD-CR, used for different purposes. PoD-AS supports fastest data delivery with $O(n)$ on-chain computation. PoD-AS* is like ZKCP (zero-knowledge contingent payment), using zkSNARKs to reduce on-chain computation to $O(1)$, but with slower off-chain delivery. Following the approach of Fairswap, PoD-CR supports fast data delivery and small on-chain computation $O(log(n))$.

We explain a prototype system – zkPoD, a practical data exchange system based on the PoD protocols and Ethereum, a popular permissionless blockchain. The system can support delivering large data file up to many GBs, and the computation complexity is quite efficient for participants with ordinary PCs. Unsurprisingly, the PoD protocols can be used widely for many data-centered scenarios, like data storage, data querying, data processing and data distributing *etc.*, achieving fairness for users while protecting their privacy. Our work also shows that blockchains can be used as trustless third parties, or verifiers, for building better protocols or systems, as well as expanding the possibility of decentralized applications.

1 INTRODUCTION

Building a protocol for trading digital commodities fairly and efficiently has long stood as a grail for Internet enthusiasts. It has formally been proved that it is impossible to achieve fairness between just two parties [18, 26, 32], (see Fig 1 (a)). A third party is needed to resolve the dilemma where the buyer is unwilling to pay prior to receiving the commodity, while the seller is indisposed to deliver until the payment is done. To break such deadlock, the third party can withhold the payment until the buyer confirms the receipt.

There have been many works about how to build three-party *fair*-exchange protocols since the 90s [1–3, 14, 17, 24]. Unfortunately, most of them impose an extra requirement on the third party: it has to be not only functional, but also trustworthy, and therein lies the problem.

The past two decades have seen a trend of centralization in the development of trusted third parties (TTPs). For ordinary users, placing their trust upon fewer entities may suggest less risk, as one of the most important principles of security states: a smaller trusted base implies a more secure system. Trusting only one TTP was apparently the most secure choice at the beginning of the Internet. But as TTPs have grown into giants blanketing most of online activities and gathering massive personal data, the risk of trusting them inevitably soars. In recent years, we have witnessed numerous security issues arising from TTP-involved privacy theft [7, 22, 27, 28, 31, 34]. They turn

Authors' address: Yuguang Hu, hyg@secbit.io; X.M. Sun, suxm@secbit.io; Yu Guo, yu.guo@secbit.io; Zhipeng Sun, zp@secbit.io; Yiwen Lu, even@secbit.io; Shaofan Wang, nio@secbit.io; Cun Ye, cunye.snd@gmail.com; Yun Li, liyunscss@gmail.com; Chao Zhang, gausszhch@gmail.com.

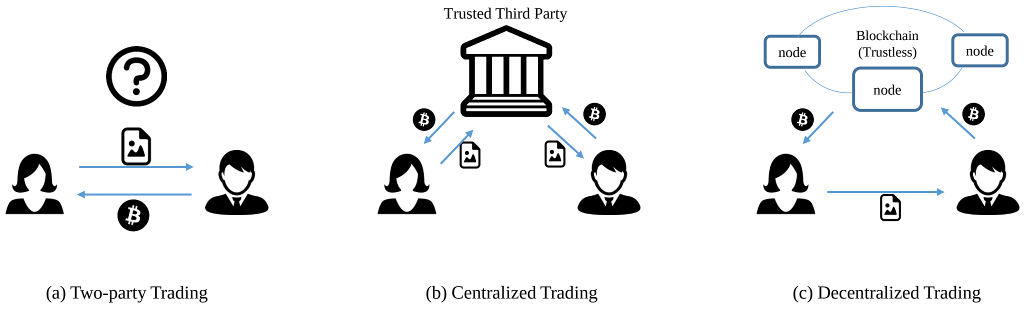


Fig. 1. Trading Patterns

out to be neither reliable nor trustworthy after all. They carry with them intrinsic weaknesses, including:

- Single-point failure risks;
- Attack threats and data leaks;
- Personal data abuse.

The technology of Blockchain [25, 35] demonstrates the possibility of building trustless applications on peer-2-peer network without any centralized authorities (Fig. 1 (c)). A few research works tackle the fair-trading problem utilizing blockchains [6, 11, 15, 23]. Instead of relying on TTPs, these works use blockchain equipped with computing scripts (smart contracts) as a *trustless* third party. The blockchain-based third party (BTP) has some significant advantages over its centralized counterpart. First, its behavior is predictable and trustworthy, since the scripts are running on every blockchain node, with the entire internal states visible to all outsiders. Moreover, the blockchain network is highly fault-tolerant that it is unlikely for the BTP to suffer a functional failure. Thirdly, since the data is transmitted off-chain, no miner can learn any information about it. Finally, it is possible to protect users' personal information by techniques like confidential transactions, anonymity protection, *etc.*, and hence avoid privacy abuse.

One approach [11, 23] relies on zkSNARKs to construct zero-knowledge proofs, putting heavy computational burden on the provers to build large-scale arithmetic circuits. Alternatively, the work of *proofs of misbehaviors* proposes a claim-or-refund strategy [6, 15]. Instead of proving its own integrity, an honest party is supposed to catch and prove to BTP the other's misbehavior. Such protocols lower the computational complexity, but impose an extra "appeal period", when the trading process halts for the participants to complain. These protocols are highly innovative, but still somewhat inadequate for practical uses.

In this paper, we propose an efficient BTP-based fair data exchange approach, named Proof of Delivery (PoD). In particular, PoD supports *atomic-swap* like ZKCP [11, 23] conducting delivery and payment in one atomic operation. We devise two variations which differ in the on-chain computational complexity. One with $O(n)$ on-chain complexity, PoD-AS, is suitable for permissioned blockchains, which facilitates fast data delivery at the rate of 3.9MiB/s. Another with $O(1)$ on-chain complexity is tailored for permissionless blockchains where throughput are limited, about 35KiB/s. It shifts most of the computations off-chain, but as a tradeoff, the buyers are required to do extra verification to ensure its security.

PoD also supports an approach of *claim-or-refund* inspired by Fairswap [6, 15]. In this approach, the trading is performed in an optimized manner where the buyer is required to submit only a very light-weighted proof showing whether the seller is cheating or not. The claim-or-refund approach amounts to a higher delivery rate of 3.3MB/s, and its computational complexity approximates to $O(log(n))$.

PoD is highly flexible, adapting to various scenarios. It can support partial delivery, where the seller can send any part of data without re-initialization. It is also possible to trade query results, *i.e.,* the buyer may pay for queries saying if some keyword exists.

In theory, PoD can integrate with any blockchain which supports smart contracts. Here we present an Ethereum-based prototype implementation – zkPoD. Any zkPoD nodes can act as either buyers or sellers. They are connected off-chain via secure channels, and interact on-chain with a smart contract, practically the judge. A seller node processes some data, generates a tag (meta-information), and publishes it to the smart contract. Then a buyer node may initiate a purchase for either a subset or the whole of the target dataset. The two parties operate interactively and invoke the smart contract following a PoD protocol. In the end if neither party cheats, data will be delivered from the seller to the buyer, and crypto-coins will be transferred from the buyer to the seller.

The zkPoD system aims to support practical data exchange. The core library is written in C++ and highly optimized. The performance is efficient enough for exchange large bulk of data. Our experimental results on a 6-core machine show that: the throughput of data pre-processing is about 11MiB/s on average, the throughputs of data delivery in PoD-AS and PoD-CR are more than 3MiB/s on average. The protocols of PoD-AS* and PoD-CR cost small gas, less than 200,000 per transaction (about 0.12 USD for a gas price of 2 GWei and 300USD/Ether) even when the size delivering data is more than 1TiB. There is still quite some room for further improvement, and we will continue working on it. Simple and flexible as PoD is, we believe it might serve as an infrastructural protocol for the next generation Internet, where information and value flow simultaneously.

2 BLOCKCHAIN-BASED FAIR EXCHANGING

Bitcoin [25] and blockchains bring a new paradigm, building *decentralized* applications, to validate digital transactions globally and instantly without relying on any central authorities – TTPs. Ethereum [35] later developed a powerful decentralized system allowing executing arbitrary user-defined programs, called smart contract [10], on the blockchain. Both Bitcoin transactions and smart contracts on Ethereum are trustworthy because they are public and transparent. Blockchain is a distributed system composed of many peer-2-peer nodes, which run a program and store results by a consensus protocol. The trust of blockchain comes from the capability of Byzantine Fault Tolerance, *i.e.,* small amount of malicious nodes are unable to interfere with honest nodes which are keeping a consistent global state.

There raises a question: if we can find a solution to replace TTPs with decentralized blockchain applications to achieve collaboration, including fair trading, then the risks brought by TTPs can be eliminated. The solution, if it exists, would solve the problems aforementioned. As we all expect, the blockchain might probably be an ideal "third party" that is *trustless*. The jobs of TTPs can be done by the blockchain or smart contracts on the blockchain, such that any behavior of a blockchain-based third party (BTP) can be checked by anyone. BTPs play the roles of "public verifiers", a concept from security protocols, which is an algorithm to verify if proofs are correct without accessing to the secrets. But the BTPs are more than public verifiers since all of the interactions of BTPs would be recorded on the blockchain. Moreover, BTPs are much more flexible to store persistent values or transfer values.

G. Maxwell and S. Bowe demonstrated a strong fair protocol ZKCP [23] for data trading. Same with the two protocols above, the seller encrypts the data with a secret K and sends encrypted data and the hash value of the secret Y to the buyer. Besides, the seller must publish zero-knowledge proofs to show that: (1) the data D is what the buyer wants; (2) the data D is correctly encrypted by K; (3) the hash computation $Hash(K) = Y$ is correct for the same K. The proofs can guarantee that the hash pre-image of Y is exactly the secret, with which the verified data can be decrypted. Then the buyer desposits bitcoins into a payment script, which can be redeemed if the seller reveals the hash pre-image of Y. We can see that the seller exchanges the key with the bitcoins simuntanously, or in an atomic way. The atomicity ensures that the seller should only reveal the correct key to get the payment. For the buyer, he knows the data is correct and he will get the key to decrypt the data if he shows the correct hash pre-image. For both parties, the protocol is fair. The protocol heavily depends on zkSNARK [5, 19, 29], a general solution for zero-knowledge proofs. However, the cost of generating proofs is considerably high. The demo might be acceptable for trading simple sudoku solutions but not practical for the data with larger size.

As pointed out by M. Campanelli *et al.*, there are vulneralbilities [11] in the experiments ZKCP conducted where a malicious verifier might extract information of Sudoku solutions by modifying the common reference strings of zkSNARK. They have also proposed a variant of ZKCP−ZKCSP that supports digital service payments. In ZKCP, a proof showing data is valid was sent to the buyer before payment, while in ZKCSP, the seller sells the "proof" to the buyer and the payment must be exchanged atomicly, before the buyer learns the knowledge proved by the proof. The approach of ZKCSP shares the same issue with ZKCP, the unaffordable cost of generating proofs on the seller's side.

Another exciting research result, Fairswap, proposed by S. Dziembowski, L. Eckey and S. Faust [15], adopts the idea of *proofs of misbehavior*, where coins will be claimed by the buyer if he finds the key is incorrect [6]. Fairswap solves the issues in ZKCP and ZKCSP, cutting off the step of generating zero-knowledge proofs showing the validity of the data. Instead, it uses a new method to show that the encryped data is valid by combining techniques of Merkle tree and abstract circuits. Thus the computation cost of the seller is reduced dramatically. The seller reveals the key directly, and then the buyer must check the key and data immediately within a time window. The buyer can generate a misbehavior proof that only includes the wrong data piece with its Merkle path, so the complexity of the proof is $O(log(n))$. The judge (the smart contract) verifies the proof with the commitments submitted by the seller. If the proof is valid, the judge refunds the buyer's payment. Otherwise, the payment is transferred to the seller. The protocol of Fairswap is optimal in the sense that if the seller is honest, the judge won't do the verification. However, if the seller reveals an incorrect key, the buyer needs to propose a proof to show that the seller is cheating. Thus the cost of on-chain verification can be very small if both parties are honest. As we can see, the pattern of Fairswap isn't atomic like ZKCP. The buyer has to stay online and claim misbehavior proofs within a predefined time window when the data is fake. Otherwise, the judge (on the blockchain) would transfer the coins deposited (by the buyer) to the seller after the time window is closed, even if the seller is cheating. The encryption scheme they used can be any CCA-secure encryption. The smart idea of proofs of misbehavior is based on computation upon abstract circuits. All of the intermediate values during the computation are encrypted block by block, and then all encrypted blocks are hashed into a Merkle tree root. If the key revealed by the seller is incorrect, it implies that there must exist one gate where the computation is incorrect. Therefore, the buyer might claim the proof by composing both the inputs and the outputs of that gate, and their Merkle paths. The judge, implemented as a smart contract, decrypts the inputs and outputs and checks the computation on the gate by recomputing them.

2.1 Our approach: PoD protocols

We take a different approach to achieve fair exchange, by an extended Schnorr protocol and Pedersen commitments. The main advantage of our PoD protocols is that the techniques we use are efficient

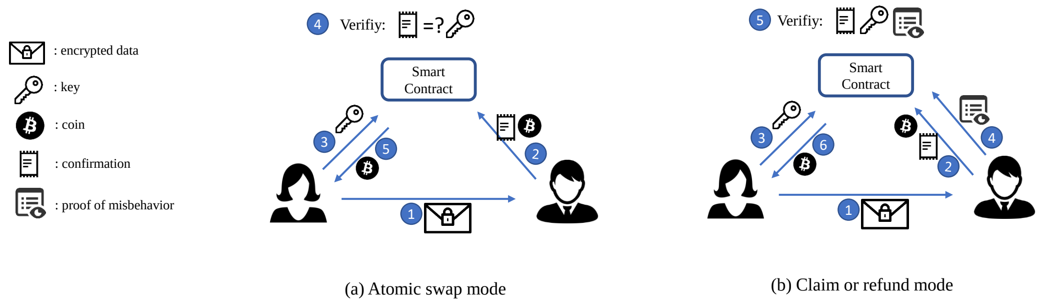


Fig. 2. Two Modes of PoD Protocols

and extensible. The protocols support atomic-swap mode (like ZKCP/ZKCSP) and claim-or-refund mode (like Fairswap) both. Although in the atomic-swap mode, the computation cost limits the speed of data delivery, the PoD protocol can support data with large size (many GBs) in one single transaction. Here we informally illustrate the protocol in a minimal setting, to let readers quickly get the basic idea of our approach.

In the PoD protocol, there are three parties: Alice as a data sender, Bob, as a data receiver and Julia as the judge. Please keep in mind that Julia is the trustless third party (implemented on the blockchain) with visible internal states and predictable behaviors. There are three phases of PoD, *Init-phase*, *Deliver-phase*, and *reveal-phase*. Suppose the data file that is going to be delivered is denoted by m and k is to denote the key encrypting the data.

We first show the atomic-swap mode of the PoD protocol, as shown in Fig. 2-(a):

(1) Init-phase: All three parties setup with system parameters. Alice publishes the *authenticator* of data (σ) by sending it to Julia such that everyone including outsiders can see it.
(2) Deliver-phase: Alice encrypts the data m to get $\overline{m}$ with a random one-time key k before she sends $\overline{m}$ and the commitment of k to Bob, who verifies two facts: (i) the encrypted data is consistent with the authenticator σ; and (ii) the data is indeed encrypted by k. If Bob accepts $\overline{m}$, he submits a *delivery receipt* containing the information of k. If the protocol is used for data trading, Bob is supposed to deposit coins at this step.
(3) Reveal-phase: After confirming the receipt, Alice reveals the key k to Julia to redeem the coins. Julia finally verifies if the key matches the delivery receipt (remember that there are information of the key in the receipt). If so, Julia accepts the key. Otherwise, Julia rejects it. If the protocol is for trading, Julia may transfer the coins, deposited by Bob, to Alice if the key is accepted. Otherwise, she returns the coins to Bob.

In the protocol, Bob's action of depositing coins isn't necessary, and Julia may invoke any smart contracts when she accepts the key, *e.g.*, writing a ledger, or informing other parties.

We can see that the protocol supports *atomic swap*, which implies that the actions of "obtaining data" and "accepting proof" are indivisible. Either they are both complete or failed. Or, the swap of "coins" and "data" is either complete, both parties get what they want; or the swap is failed, neither of the two parties get anything. We will present two variants of atomic-swap mode of PoD in Sec. 3. One of which, named PoD-AS*, is similar to ZKCP, where the Julia only has to verify a hash pre-image by using zkSNARKs to build zero-knowledge proofs. But PoD also supports a variant (PoD-AS) to lower the computation burden of Alice by avoiding build proofs of zkSNARK. The variant is suitable for permissioned blockchain, where the performance (TPS) is high and computation cost of smart contract is pretty low.

The claim-or-refund mode, presented in Fig. 2-(b), adopts the approach from Fairswap, realizing fair trading by using so-called *proofs of misbehavior*. It requires Bob to submit the evidence (proofs) to Julia if he does find that Alice is cheating, *i.e.,* showing a wrong key or sending wrong data. Julia checks the proof claimed by Bob and decides if to punish Alice. The big advantage of Fairswap is that the computation complexity of proofs of misbehavior is only $O(log(n))$. In this mode, the protocol also has three phases:

(1) Init-phase: All three parties do setup and Alice computes and publishes the *authenticator* of data, σ.
(2) Deliver-phase: Alice sends K, the commitment to k, to Bob, as well as the encrypted data into $\overline{m}$ with k. Bob verifies two facts: (i) the encrypted data is consistent with the authenticator σ; and (ii) the data is indeed encrypted by some key whose commitment is K. If Bob accepts $\overline{m}$, he submits a *delivery receipt* containing the information of K. If the protocol is used for trading, Bob is supposed to deposit coins at this step.
(3) Reveal-phase: Alice reveals k to Julia, who creates a timer and waits for Bob to complain. If k is correct, Bob does nothing, and Alice will be able to withdraw the coins deposited by Bob. If k is incorrect, Bob must submit a proof showing that the key is wrong within a time limit. Julia verifies the proof and transfers the coins to Alice if the proof is invalid. Otherwise, the coins would be returned to Bob.

PoD protocols are built on some cryptographic primitives, and we briefly explain them here.

Homomorphic authenticator. In data trading, Bob's (buyer) concern is if the data is what he wants. The PoD protocols uses homomorphic authenticators [33] as the meta-data such that Bob can confirm the origin and integrity of the data before they are decrypted. The authenticators are generated from the raw data but don't leak any information to be learned by adversaries. The authenticators should be published after initialization, and anyone can access to them. We use Pedersen commitments [30] to construct data authenticators, which are perfectly hiding and computationally binding. Besides, the Pedersen commitments are also additively homomorphic, which means that multiplying two commitments produces a commitment to the sum of the openings.

Off-chain and on-chain verification. If Bob is honest but Alice is possibly malicious, Bob has to make sure two points: (i) the data are what he wants, and (ii) the key revealed is exactly the key used to decrypt the data. The verification is divided into two parts: the verification done by Bob, and the verification done by Julia (in public). The former is done off-chain, such that any miners or outsiders cannot learn anything about data. The latter should be done by public verifiers such that Julia may be aware of the data delivery between Alice and Bob. Therefore, Bob should provide Julia sufficient information about the key for key validation, *e.g.,* the commitment of the key. That's why Bob needs to submit a delivery receipt to Julia. Bob verifies two things: (i) the (encrypted) data is corresponding to the authenticators; (ii) the key of (encrypted) is the opening to the commitment sent from Alice. The verification on-chain of Julia ensures one thing: the key should be correct. There is another subtle question – what if Bob submits a fake receipt? If Bob cheated, Alice wouldn't get the coin after revealing the real key. Thus Alice must also verify that the receipt from Bob is correct. The delivery-receipt doesn't have to be sent to Alice since Julia's internal states are visible to everyone, including Alice.

Fairness and zero-knowledge. Suppose Alice is honest and her concern is that Bob might be malicious. If Bob quits after he receives the (encrypted) data, he might try to extract valuable information from them. The protocol has to ensure that malicious Bob cannot learn any more information about the data except the fact that the data is valid. In other words, the verification of Bob has to be zero-knowledge [16]. PoD protocol uses a variant of $\Sigma-$protocol to ensure honest verifier zero-knowledge and it can be turned into NIZK by using techniques from Fiat-Shamir Heuristic [13].

Encryption with one-time pad. Because Alice reveals the key to Julia, Alice has to use a randomly choosen one-time key for each transaction. The encryption scheme we use is a one-time pad, in which each data piece is simply added by the key. That implies that the size of keys is equal to the size of data. With a one-time pad, it is easy to make sure that the data is correctly encrypted with some key through the homomorphic property of Pedersen commitments. However, if the data delivered is large, the number of keys will be large. It would lead to high communication complexity $O(n)$. To solve the problem, the PoD protocol uses a hash function to generate all one-time keys from a single seed. Also, the PoD protocol uses a linear combination to encode keys to generate the delivery receipt to reduce the computation on Julia, since computation on finite fields cost less than ECC exponentiations.

2.2 zkPoD: a practical data exchange system based on Ethereum

Based on PoD protocols, we build a system for data trading – zkPoD. The system contains nodes and smart contracts. One node is a client program running on the users' side, while smart contracts are deployed on Ethereum. Nodes can communicate with each other through TCP/IP connections, P2P sharing networks, or IPFS network. The system is quite efficient and practical, supporting data delivery with large data size.

Decentralization and availability. Any user may share data with others without registration. All they have to do before using the system is to create a key-pair of Ethereum, which consists of thousands of peer nodes globally and can hardly be devastated. The main advantage of decentralization is that no one can stop anyone from using the system.

Fairness and trustworthy. The trading protocol is strong fair, *i.e.,* if the protocol ends, neither party (the buyer or the seller) will have more advantages over the other side. Moreover, if any party aborts at any step of the protocol, the property of fairness still holds.

Privacy preserving. The system is privacy-friendly in design. No information of users is required. Any miner of the blockchain can obtain any piece of data, since all data are transferred off-chain. The intention of buyers can be hidden by using techniques of oblivious transfer.

Blockchain independence The system can deploy the code of Julia on different blockchains with smart contracts apart from Ethereum. The algorithm of the public verifier, Julia, is rather lightweight and easy to implement. Unsurprisingly, it is independent of the consensus protocol of the blockchain.

Efficiency. The performance of zkPoD is good enough for data delivery with many GBs. The throughput of data is up to 3 MB/s. The heaviest computation burden on users is exponentiations on ECC groups. The experimental results show it takes only roughly 330s to deliver data with the size of 1G, with 200,000 gas only.

3 POD PROTOCOLS

The goal of PoD protocols is to realize public verifiable data delivery with a blockchain-based third party (BTP), who ensures that the delivery process is trustworthy. It is obvious that the problem of fair trading can be solved by the PoD protocol when cryptocurrencies are added. The protocol adopts some basic cryptographic tools, *e.g.,* Pedersen commitments, Schnorr protocol and zero-knowledge proof construction.

In this section, we explain the principle of PoD by presenting a minimal version of protocol which supports data delivery of a few bytes only. But it is easy for readers to get the ideas.

3.1 Preliminaries

In figures, we use $A \xrightarrow{x} B$ to denote that A sends x to B in a *secure* communication channel. We use $B \xRightarrow{y} C$ to denote that B *broadcasts* y to C publicly while others can be notified of the event and y can be learned by all parties in the environment.

Definition 3.1 (Discrete Logarithm Assumption). The discrete logarithm assumption holds for $\mathbb{G} = \langle g \rangle$ with a prime order p if for all non-uniform polynomial time adversaries $\mathcal{A}$ such that:

$$\Pr\left[\ g^a = h \ \middle|\ h \overset{\$}{\leftarrow} \mathbb{Z}_p; a \leftarrow \mathcal{A}(\mathbb{G}, g, h)\ \right] \approx 0$$

Lemma 3.2 (Schwartz-Zippel). *Let poly be a non-zero multivariate polynormial of degree d over $\mathbb{Z}_p$, then the probability of $poly(x_1, \ldots, x_m) = 0$ for randomly chosen $x_1, \ldots, x_m \overset{\$}{\leftarrow} \mathbb{Z}_p$ is at most d/p.*

Definition 3.3 (Pedersen Commitment). A Pedersen commitment scheme is a pair of probabilistic polynomial time algorithms $(\mathsf{Gen}, \mathsf{Com})$ such that:

- $\mathsf{Gen}(1^\lambda)$: The algorithm chooses two random generators $g \in \mathbb{G}_1$, and $h \in \mathbb{G}_1$; and outputs a commitment key: $\mathsf{ck} = (g, h)$.
- $\mathsf{Com}_{\mathsf{ck}}(m; r)$: Given a message m, and a randomness r, outputs a commitment to m:

$$\mathsf{Com}_{(g,h)}(m; r) = g^m \cdot h^r$$

We say a commitment scheme is additively homomorphic if for all valid keys ck, and for all messages $m_1, m_2 \in \mathbb{Z}_p$ and randomness $r_1, r_2 \in \mathbb{Z}_p$, we have

$$\mathsf{Com}_{\mathsf{ck}}(m_1; r_1) \cdot \mathsf{Com}_{\mathsf{ck}}(m_2; r_2) = \mathsf{Com}_{\mathsf{ck}}(m_1 + m_2; r_1 + r_2)$$

Theorem 3.4 (Commitment Perfect Hiding). *The commitment scheme of (Gen, Com) is perfect hiding if for all probabilistic polynomial time stateful interactive adversaries $\mathcal{A}$ such that:*

$$\Pr\left[\ \mathcal{A}(c) = b \ \middle|\ \begin{array}{l} (m_0, m_1) \leftarrow \mathcal{A}(ck); b \leftarrow \{0,1\}; \\ r \overset{\$}{\leftarrow} \mathbb{Z}_p; c \leftarrow Com(m_b; r) \end{array}\ \right] = \frac{1}{2}$$

where $\mathcal{A}$ outputs $m_0, m_1 \in \mathbb{Z}_p$.

Definition 3.5 (Commitment Computational Binding). The commitment scheme of data blocks is computationlly binding if for all probabilistic polynomial time adversaries $\mathcal{A}$, such that:

$$\Pr\left[\ \begin{array}{c} \mathsf{Com}(m_1; r_1) = \mathsf{Com}(m_2; r_2) \\ m_1 \neq m_2 \end{array} \ \middle|\ (m_1, r_1, m_2, r_2) \leftarrow \mathcal{A}(\mathsf{ck})\ \right] \approx 0$$

where $\mathcal{A}$ outputs $m_0, m_1 \in M$.

Definition 3.6 (MerkleTree). A MerkleTree scheme is a triple of probabilistic polynomial time algorithms $(MKRoot, MKPath, MKVerify)$ such that:

- $MKRoot(x_1, \ldots, x_t)$: The algorithm takes a list of numbers $x_1, \ldots, x_t$; and outputs a value in $\mathbb{Z}_p$:

$$\gamma = MKRoot(x_1, \ldots, x_t)$$

- $MKPath(\gamma, i, x_1, \ldots, x_t)$: Given a Merkle root γ, an index of the element i, and the list of all members $x_1, \ldots, x_t$; the algorithm outputs a Merkle proof of x_i, ξ:

$$\xi \leftarrow MKPath(\gamma, i, x_1, \ldots, x_t)$$

- $MKVerify(\gamma, x_i, \xi, l)$: Given a Merkle root γ, an element x_i, , a proof ξ and the length of set l; the algorithm outputs 1 if the element exists in the Merkle tree, and 0 otherwise.

$$\{1, 0\} \leftarrow MKVerify(\gamma, x_i, \xi, l)$$

Definition 3.7 (Random oracle). A random oracle is a function of probabilistic polynomial time algorithms H such that:

- $H(x_1, \ldots, x_q)$: The algorithm takes a arithmetic circuit C and security

$$h \leftarrow H(x_1, \ldots, x_q) = hash(x_1 \parallel \cdots \parallel x_q)$$

LEMMA 3.8 (RANDOM ORACLE ADDITION). *Let H be a random oracle with uniform distribution, then the function H^* is also a random oracle with uniform distribution for any $k_1, k_2, \ldots, k_n$:*

$$H^*(x) \leftarrow H(x, k_1) + H(x, k_2) + \cdots + H(x, k_n)$$

where $(+)$ is a modular addition.

Definition 3.9 (zkSNARK). A zkSNARK scheme is a triple of probabilistic polynomial time algorithms (zkSetup, zkProve, zkVerify) such that:

- zkSetup$(C, 1^\lambda)$: The algorithm takes a circuit, and security parameter λ; and outputs a pair of keys, ek and vk:

$$(ek, vk) = \mathsf{zkSetup}(C, 1^\lambda),$$

- zkProve$_{ek}(\vec{x}, \vec{y}, \vec{w})$: Given public circuit inputs $\vec{x}$, circuit outputs $\vec{y}$, and private circuit inputs $\vec{w}$; the algorithm outputs an argument (zkSNARK):

$$\pi \leftarrow \mathsf{zkProve}_{ek}(\vec{x}, \vec{y}, \vec{w})$$

- zkVerify$_{vk}(\vec{x}, \vec{y}, \pi)$: public circuit inputs $\vec{x}$, circuit outputs $\vec{y}$, and an argument π; the algorithm outputs 1 if the argument is valid, and 0 otherwise.

$$\{1, 0\} \leftarrow \mathsf{zkVerify}_{vk}(\vec{x}, \vec{y}, \pi)$$

3.2 PoD-Mini: minimized proof of delivery

The minimal PoD protocol has three parties, a sender, a receiver and a judge. It has three phases, aformentioned in Sec 2.1, init-phase, deliver-phase and reveal-phase. The protocol is shown in Fig 3.

Definition 3.10. PoD-Mini protocol. A PoD-Mini protocol is a tuple $(\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{J})$ of probalistic polynomial time algorithms such that

- $\mathcal{I}(m)$ is an algorithm called ***initializer***, which takes as input data/message and outputs data with a padding block and group elements to $\mathcal{S}$; it also outputs an authenticator and group elements to $\mathcal{R}$;
- $\mathcal{S}(m, o, g, h)$ is an interactive protocol algorithm called ***sender***, which takes as input data/message, a padding block, and two group elements, and sends encrypted data with other information to $\mathcal{R}$, and it also outputs keys to $\mathcal{J}$.
- $\mathcal{R}(\sigma, g, h)$ is an interactive protocol algorithm called ***receiver***, which takes as input the authenticators and group elements to verify the encrypted data, and outputs a receipt to $\mathcal{J}$.
- $\mathcal{J}(g, h)$ is an interactive protocol algorithm called ***Judge***, which takes as input the receipt from $\mathcal{R}$ and the keys from $\mathcal{S}$ to verify the keys, and outputs 1 or 0 to accept or reject the proof.

In the init-phase, $\mathcal{I}()$ is invoked to setup system parameters and initialize m, which is used to denote the data that will be delivered to $\mathcal{R}$. The We use Pedersen commitments to hide m with randomly picked data piece o:

$$\sigma = \mathsf{Com}(m; o) = g^m \cdot h^o;$$

and Com$(m; o)$ is used to denote a commitment and $m, o \in \mathbb{Z}_p$ are converted into finit field elements from strings of bits, assuming data are small. We call the commiment of a data block as an authenticator, denoted by σ. In PoD-Mini, σ can only be used to check data integrity. By the property of *computationally binding* of Pedersen commitment scheme, the commitment Com$(m; o)$ of m can be

$$\underline{\mathcal{I}(m)}$$

$$g, h \xleftarrow{\$} \mathbb{G}$$

$$\sigma = g^m \cdot h^o; o \xleftarrow{\$} \mathbb{Z}_p$$

$\underline{\mathcal{S}(m, o, g, h)}$ $\qquad\qquad$ $\underline{\mathcal{R}(\sigma, g, h)}$

$$k_\omega \xleftarrow{\$} \mathbb{Z}_p$$

$$k \leftarrow H(k_\omega, 1); k' \leftarrow H(k_\omega, 2)$$

$$K = g^k h^{k'};$$

$$k_0 \leftarrow H(k_\omega, 3); k_0' \leftarrow H(k_\omega, 4)$$

$$K_0 = g^{k_0} h^{k_0'};$$

$$\xrightarrow{K_0, K}$$

$$c \xleftarrow{\$} \mathbb{Z}_p$$

$$\xleftarrow{c}$$

$$\overline{m} = k + c \cdot m; \overline{o} = k' + c \cdot o$$

$$z = k_0 + c \cdot k; z' = k_0' + c \cdot k'$$

$$\xrightarrow{(\overline{m}, \overline{o}, z, z')}$$

$$\sigma^c \cdot K \stackrel{?}{=} g^{\overline{m}} h^{\overline{o}}$$

$$K_0 \cdot K^c \stackrel{?}{=} g^z h^{z'}$$

$$\underline{\mathcal{J}(g, h)}$$

$$(z, c) \stackrel{?}{=} \varrho \qquad \xLeftarrow{\text{receipt:}\varrho=(z,c)}$$

$$\xRightarrow{\text{revealing:}k_\omega} \qquad m = (\overline{m} - H(k_\omega, 1))/c$$

$$\varrho.z \stackrel{?}{=} H(k_\omega, 3) + \varrho.c \cdot H(k_\omega, 1)$$

Fig. 3. PoD-Mini Protocol

published used as the tag of the data. $\mathcal{R}$ can easily verify the opening, (m, o), to know if the data received is correct.

In the deliver-phase, $\mathcal{S}$ and $\mathcal{R}$ conduct a three-move interaction. First, $\mathcal{S}$ generates two random keys k, k' in $\mathbb{Z}_p$ by using the random oracle function $H(\cdot)$, such that k can be hidden in a commitment, K_1:

$$K = \mathsf{Com}(k; k') = g^k \cdot h^{k'}.$$

$\mathcal{S}$ also picks two additional random numbers k_0, k_0' by the random oracle $H(\cdot)$ to compute another commitment, which is used to prove zero-knowledge.

$$K_0 = \mathsf{Com}(k_0; k_0') = g^{k_0} \cdot h^{k_0'}.$$

In the first move, $\mathcal{S}$ sends two commitments K_0, K to $\mathcal{R}$. And then $\mathcal{R}$ returns a random challenge number $c \in \mathbb{Z}_p$ to $\mathcal{S}$ in the second move. In the third move, $\mathcal{S}$ encrypts m and o using one-time pad with the two keys k, k'. Please note here k is for m and k' is for o (the padding block).

$$\overline{m} = k + c \cdot m, \qquad \overline{o} = k' + c \cdot o.$$

Also $\mathcal{S}$ hides the two keys by two random numbers generated in the first move.

$$z = k_0 + c \cdot k, \qquad z' = k'_0 + c \cdot k'.$$

Then $\mathcal{S}$ sends $(\overline{m}, \overline{o}, z, z')$ to $\mathcal{R}$, who verifies if the encrypted data is correct by using the homomorphic property of Pedersen commitments:

$$\mathsf{Com}(m; o)^c \cdot \mathsf{Com}(k; k') \overset{?}{=} \mathsf{Com}(\overline{m}; \overline{o}).$$

From the equation above, R knows that (1) the plaintext of $\overline{m}$, saying m, is bound to $\mathsf{Com}(m; o)$; and (2) there exists a secret key k, bound to $\mathsf{Com}(k; k')$, used for encryption. $\mathcal{R}$ also verifies if the sender $\mathcal{S}$ actually has the knowledge of (k, k') by check the following equation:

$$\mathsf{Com}(k_0; k'_0) \cdot \mathsf{Com}(k; k')^c \overset{?}{=} \mathsf{Com}(z; z')$$

If $\mathcal{R}$ accepts the data, she will go to the next phase, otherwise she aborts. We can see that k and k' are for encryption, and k_0 and k'_0 are random numbers for protecting k and k'. The interaction of the deliver-phase can be proved zero-knowledge and knowledge soundness. That's to say, we can construct an extractor interacting with a cheating sender to exact the knowledge; and also, a simulator can be built to cheat an honest receiver with public coins.

In the reveal-phase, $\mathcal{R}$ and $\mathcal{S}$ interact with $\mathcal{J}$ to prove that the data is delivered (or not). If $\mathcal{R}$ accepts the information sent from $\mathcal{S}$, she submits to $\mathcal{J}$ a delivery-receipt, $\rho = z_0$, which is the encoding of k, k_0. Then $\mathcal{S}$ needs to reveal k_ω to $\mathcal{J}$ if ρ is equal to z_0. If the receipt is incorrect, $\mathcal{S}$ aborts. Finally, $\mathcal{J}$ verifies the revealed k_ω by checking the follow equation:

$$z_0 \overset{?}{=} H(k_\omega, 3) + c \cdot H(k_\omega, 1)$$

Definition 3.11 (Relation-of-Delivery). A relation is a polynomial-time-decidable set R, such that:

$$R = \{((\mathsf{ck}, \sigma, \overline{m}, z; \rho), m) : \exists\, o\,.\, \sigma = \mathsf{Com}_{\mathsf{ck}}(m; o) \ \wedge\ \exists\, k\,.\, \mathit{Decrypt}(\overline{m}, \rho, k) = m \ \wedge\ \exists\, k_0\,.\, z = \mathit{Encode}(k, \rho; k_0)\}$$

Definition 3.12 (Perfect Completeness of Deliver-phase). The protocol $(\mathcal{I}, \mathcal{S}, \mathcal{R})$ has perfect completeness if for all polynomial-time adversaries $\mathcal{A}$,

$$\Pr\left[\ \mathcal{R}(\sigma, K_0, K_1, c, \overline{m}, z, z') = 1 \ \middle|\ \begin{array}{l} (\mathsf{ck}, m, o, \sigma) \leftarrow \mathcal{A}(1^\lambda); \\ (K_0, K_1, \overline{m}, z, z') \leftarrow \mathcal{S}(\mathsf{ck}, m, o); c \xleftarrow{\$} \mathbb{Z}_p; \\ (\mathsf{ck}, \sigma, \overline{m}, z; m, k) \in R(\rho) \end{array} \right] = 1$$

Definition 3.13 (Perfect Public-coin Special HVZK of Deliver-phase). The protocol $(\mathcal{I}, \mathcal{S}, \mathcal{R})$ is special honest verifier zero knowledge, or sHVZK, if there exists an efficient probabilistic algorithm $\mathcal{E}$ (called a simulator) that can output indistinguishable interactions with the receiver only given public inputs.

$$\Pr\left[\begin{array}{l} (\mathsf{ck}, \sigma, \overline{m}, z; m) \in R(\rho) \\ \mathcal{A}(K_0, K, \rho, \overline{m}, z, z') = 1 \end{array} \ \middle|\ \begin{array}{l} m \leftarrow \mathcal{A}(1^\lambda); (\mathsf{ck}, o, \sigma) \leftarrow \mathcal{I}(m) \\ (K_0, K, \rho, \overline{m}, z, z') \leftarrow \langle \mathcal{S}(\mathsf{ck}, m, o), \rho, \mathcal{R}(\mathsf{ck}, \sigma) \rangle \end{array} \right] =$$
$$\Pr\left[\begin{array}{l} (\mathsf{ck}, \sigma, \overline{m}, z; m, k) \in R(\rho) \\ \mathcal{A}(K_0, K, \rho, \overline{m}, z, z') = 1 \end{array} \ \middle|\ \begin{array}{l} m \leftarrow \mathcal{A}(1^\lambda); (\mathsf{ck}, o, \sigma) \leftarrow \mathcal{I}(m) \\ (K_0, K, \rho, \overline{m}, z, z') \leftarrow \mathcal{E}(\mathsf{ck}, \sigma; \rho) \end{array} \right]$$

where ρ is the public coin randomness used by the receiver.

Definition 3.14 (Computational Knowledge Soundness of Deliver-phase). The interactive proof system of $(\mathcal{I}, \mathcal{S}, \mathcal{R})$ is computationally knowledge sound if for all polynomial-time adversaries $\mathcal{A}$, there exists an efficient deterministic algorithm $\mathcal{X}()$ that can always output the data m which is the plaintext of the encrypted data $\overline{m}$.

$$\Pr\left[\ \mathcal{A}(tr)=1\ \middle|\ \begin{array}{l} m \leftarrow \mathcal{A}(1^\lambda); (\mathsf{ck}, o, \sigma) \leftarrow \mathcal{I}(m); \\ tr \leftarrow \langle \mathcal{S}^*(\mathsf{ck}, m, o), \rho, \mathcal{R}(\mathsf{ck}, \sigma)\rangle \\ (\mathsf{ck}, K_0, K, \rho, \overline{m}, z, z') = tr \end{array}\right] \approx$$

$$\Pr\left[\ \begin{array}{l} \mathcal{A}(tr)=1 \\ \wedge\, \mathcal{R}(\mathsf{ck}, tr) = 1 \longrightarrow (\mathsf{ck}, \sigma, \overline{m}, z; m', k) \in R(\rho) \end{array}\ \middle|\ \begin{array}{l} m \leftarrow \mathcal{A}(1^\lambda); (\mathsf{ck}, o, \sigma) \leftarrow \mathcal{I}(m); \\ (tr, m') \leftarrow \mathcal{X}^{\langle \mathcal{S}^*(\mathsf{ck}, m, o), \_, \mathcal{R}(\mathsf{ck}, \sigma)\rangle}(\mathsf{ck}, \sigma) \\ (\mathsf{ck}, K_0, K, \rho, \overline{m}, z, z') = tr \end{array}\right]$$

Definition 3.15 (Computational Binding of Reveal-phase). The interactive proof system of $(\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{J})$ is computationally binding if for all polynomial-time adversaries $\mathcal{A}$,

$$\Pr\left[\ k_\omega \neq k'_\omega\ \middle|\ \begin{array}{l} m \leftarrow \mathcal{A}(1^\lambda); (\mathsf{ck}, o, \sigma) \leftarrow \mathcal{I}(m); \\ tr \leftarrow \langle \mathcal{S}(\mathsf{ck}, m, o), \rho, \mathcal{R}(\mathsf{ck}, \sigma)\rangle \\ (\mathsf{ck}, K_0, K, \rho, \overline{m}, z, z') = tr \\ (k_\omega, k'_\omega) \leftarrow \mathcal{A}(\rho, z, z') \\ 1 \leftarrow \mathcal{J}(z, k_\omega)\ \wedge\ 1 \leftarrow \mathcal{J}(z, k'_\omega) \end{array}\right] \approx 0$$

3.3 PoD-AS: Support Atomic-swap for permissioned blockchains

The protocol PoD-Mini can only deliver a piece of data fitted in an integer in $\mathbb{Z}_p$. To support data with larger size, for example size greater than 1MB, we present an extended version, PoD-AS, which solves the following issues:

1) the size of authenticators should be compact to download, while the size of an authenticator is equal to the size of a data block in PoD-Mini;
2) the size of delivery-receipts and revealings should be compact for $\vdash$ to verify.

The basic idea of PoD-AS is that we use *Vector Pedersen Commitment* and techniques of zero-knowledge proof construction for matrices [20].

Init-phase: In the init-phase, the data file is splitted into a block matrix of $n \times s$. Each row of the matrix is called a *block*, which consists of s slices. The initializer adds one additional column of random slices m_{0i} to the matrix for padding. The slices of m_{0i} are used for blind factors as o in PoD-Mini.

$$\left[\begin{array}{c|ccccc} m_{10} & m_{11}, & m_{12}, & \ldots, & m_{1s} \\ m_{20} & m_{21}, & m_{22}, & \ldots, & m_{2s} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ m_{n0} & m_{n1}, & m_{n2}, & \ldots, & m_{ns} \end{array}\right]$$

We use notations with ranges to denote vectors, *e.g.,* $\mathbf{r}_{[1,n]}$, $\mathbf{m}_{[1,n][0,s]}$. The initializer needs to generate $n+1$ group elements randomly.

$$\mathbf{u}_{[0,s]} = (u_0, u_1, u_2, \ldots, u_s), \text{ where } u_i \overset{\$}{\leftarrow} \mathbb{G},\ i \in [0, s].$$

Then she computes n authenticators, each of which is for one row of block matrix.

$$\sigma_i = \mathsf{Com}(m_{i1}, \ldots, m_{is}; m_{i0}) = u_0^{m_{i0}} \cdot \textstyle\prod_{j=1}^{s} u_j^{m_{ij}},$$

The vector $\sigma_{[1,n]}$ is shared by the sender and the receiver. The size of the vector is only $1/s$ of the data file.

Deliver-phase: To encrypt the matrix of data, $\mathcal{S}$ needs to generate $n * (s+1)$ random numbers as one-time-pad keys. If $\mathcal{S}$ naively chooses random keys, they would be too large to reveal to $\mathcal{J}$ in the reveal-phase. Instead, the sender uses a random oracle $H(\cdot)$ to generate these keys from a seed, k_ω:

$$k_{ij} = H(k_\omega, i, j).$$

$\mathcal{S}$ can derive a matrix of keys form the seed, and the size of the matrix is $(n+1) * (s+1)$.

$$\begin{array}{c|cccc}
k_{00} & k_{01}, & k_{02}, & \ldots, & k_{0s} \\
\hline
k_{10} & k_{11}, & k_{12}, & \ldots, & k_{1s} \\
k_{20} & k_{21}, & k_{22}, & \ldots, & k_{2s} \\
\cdots & \cdots & \cdots & & \cdots \\
k_{n0} & k_{n1}, & k_{n2}, & \ldots, & k_{ns}
\end{array} .$$

Note that there are $(n + 1)$ rows and $(s + 1)$ columns of keys. The leftmost column is for encrypting padding slices, k_{i0}, while the topmost row is for hiding keys in the same column, *i.e.,* , each k_{0j} is for hiding the keys of $(k_{1j}, k_{2j}, \ldots, k_{nj})$. Then $\mathcal{S}$ constructs commitments K_i to i-th row of keys, including the leftmost key k_{i0} on each row. The vector of commitments $\mathbf{K}_{[0,n]}$ are sent to $\mathcal{R}$ and then $\mathcal{S}$ can get a challenge number c back from $\mathcal{R}$, which is supposed to be a randomly chosen number. As the last move of deliver-phase, $\mathcal{S}$ encrypts each message slice m_{ij} by k_{ij}. The sender $\mathcal{S}$ also needs to construct a vector of $\mathbf{z}_{[0,s]}$ that are arguments showing that $\mathcal{S}$ knows the openings of all commitments $\mathbf{K}_{[0,n]}$. In the third move, $\mathcal{S}$ sends $\mathcal{R}$ three components: encrypted slices $\overline{\mathbf{m}}_{[1,n][1,s]}$ and $\mathbf{z}_{[0,s]}$. Please note that all private data sent by $\mathcal{S}$ are perfectly hidden, and $\mathcal{R}$ will make sure that there are openings *computationally* bound to the commitments.

The algorithm of $\mathcal{R}$ is defined in Fig 4. After getting commitments $\mathbf{K}_{[0,n]}$ of keys from the sender, $\mathcal{R}$ picks a random number c from $\mathbb{Z}_p$ and returns it to $\mathcal{S}$. Next, $\mathcal{S}$ is supposed to send encrypted data and auxiliaries. The receiver $\mathcal{R}$ then first verifies the encrypted data by homomorphic authenticators:

$$\prod_{i=1}^{n} \sigma_i^{(c^i)} \cdot K_i \overset{?}{=} \prod_{i=1}^{n}\Big(\prod_{j=0}^{s} u_j^{\overline{m}_{ij}}\Big). \tag{1}$$

The check ensures that the encrypted data is associated to authenticators, and the one-time-pad keys are the key for encryption. $\mathcal{R}$ also needs to verify if the sender has the knowledge of the one-time-pad keys, in other words, $\mathcal{S}$ has the openings to $\mathbf{K}_{[0,n]}$. If $\mathcal{R}$ accepts the keys and data, he has to submit a delivery receipt to $\mathcal{J}$ (z, c), where z is the aggregation of $\mathbf{z}_{[0,s]}$:

$$z = \sum_{j=0}^{s} z_j \tag{2}$$

Reveal-phase: In the reveal-phase, $\mathcal{R}$ sends a receipt (z, c) to $\mathcal{J}$ if $\mathcal{R}$ accepts the data. Then $\mathcal{S}$ reveals the key k to $\mathcal{J}$ if she checks that the receipt is correct. The remaining job of $\mathcal{J}$ is to verify if the key revealed is correct:

$$z \overset{?}{=} \textstyle\sum_{i=0}^{n} \sum_{j=0}^{s} k_{ij} \cdot c^i,$$

where k_{ij} is derived from $H(k_\omega, i, j)$.

The protocol is provably *special HVZK* and has *special knowledge soundness*. The protocol here is compatible to support *partial data delivery*, in which $\mathcal{R}$ can tell $\mathcal{S}$ which blocks (rows) are interested, and then they run the deliver-phase as usual.

Also, $\mathcal{R}$ doesn't have to download all authenticators before pariticipating, if he only downloads a small portion of data. The protocol can easily support to download authenticator on demand and verify authenticators by using Merkle proofs. $\mathcal{S}$ only needs to compute a merkle tree root of $\sigma_{[1,n]}$, saying γ, and publish it. $\mathcal{S}$ can send σ_i along with their merkle proofs (merkle path) to $\mathcal{R}$ in the deliver-phase. It will be easy for $\mathcal{R}$ to verify if the authenticators are correct.

Efficiency. The communication complexity of PoD-AS is $(n \cdot s + 2n + s + 2)$, consisting of encrypted data $(n \cdot s)$, the commitments of keys $(n + 1)$, the encodings of keys s, and authenticators n. The computation of initialization is $(n \cdot s)$ exponentiations with multiplication on $\mathbb{G}$, or (n) multi-exps. The computation of proving are roughly $(n \cdot s)$ exponentiations with multiplication on $\mathbb{G}$, $(n \cdot s)$

$$\underline{\mathcal{I}(\mathbf{m}_{[1,n][1,s]})}$$

$$u_j \xleftarrow{\$} \mathbb{G},\ {}_{j\in[0,s]}$$

$$m_{i0} \xleftarrow{\$} \mathbb{Z}_p,\ {}_{i\in[1,n]}$$

$$\sigma_i = \prod_{j=0}^{s} u_j^{m_{ij}},\ {}_{i\in[1,n]}$$

$$\underline{\mathcal{S}(\mathbf{m}_{[1,n][0,s]}, \mathbf{u}_{[0,j]})} \qquad\qquad \underline{\mathcal{R}(\sigma_{[1,n]}, \mathbf{u}_{[0,s]})}$$

$$k_\omega \xleftarrow{\$} \mathbb{Z}_p;$$

$$k_{ij} \leftarrow H(k_\omega, i, j);\ {}_{i\in[1,n],j\in[0,s]}$$

$$K_i = \prod_{j=0}^{s} u_j^{k_{ij}};\ {}_{i\in[0,n]}$$

$$\xrightarrow{\mathbf{K}_{[0,n]}}$$

$$\xleftarrow{c} \qquad\qquad c \xleftarrow{\$} \mathbb{Z}_p$$

$$\overline{m}_{ij} = k_{ij} + m_{ij} \cdot c^i;\ {}_{i\in[1,n],j\in[0,s]}$$

$$z_j = \sum_{i=0}^{n} k_{ij} \cdot c^i;\ {}_{j\in[0,s]}$$

$$\xrightarrow{\overline{\mathbf{m}}_{[1,n][0,s]}, \mathbf{z}_{[0,s]}} \qquad \prod_{i=1}^{n} \sigma_i^{(c^i)} \cdot K_i \overset{?}{=} \prod_{i=1}^{n} \prod_{j=0}^{s} u_j^{\overline{m}_{ij}}$$

$$\prod_{i=0}^{n} K_i^{(c^i)} \overset{?}{=} \prod_{j=0}^{s} u_j^{z_j}$$

$$z = \sum_{j=0}^{s} z_j$$

$$\underline{\mathcal{J}()}$$

$$\xLeftarrow{\text{receipt:}\varrho=(z,c)}$$

$$\varrho \overset{?}{=} \left(\prod_{j=0}^{s} z_j, c\right)$$

$$\xRightarrow{\text{revealing:}k_\omega}$$

$$k_{ij} \leftarrow H(k_\omega, i, j);\ {}_{i\in[0,n],j\in[1,s]}$$

$$\varrho.z \overset{?}{=} \sum_{i=0}^{n} \sum_{j=0}^{s} k_{ij} \cdot (\varrho.c)^i$$

Fig. 4. PoD-AS: Atomic-swap

hashes (using a hash primitive to realize the random oracle function), and $(n \cdot s + s)$ multiplications on $\mathbb{Z}_p$. The computation of verifying are mainly $(n \cdot s + 2n + s)$ exponentiations on $\mathbb{G}$. The PoD-AS protocol avoids expensive computation on $\mathbb{G}$ in the reveal-phase. It includes $(n \cdot s)$ hashes and $(2n \cdot s)$ multiplications and $(n \cdot s)$ additions on $\mathbb{Z}_p$.

3.4 PoD-AS*: a protocol for permissionless blockchains

The PoD-AS supporting atomic swap presented in Sec. 3.3 is suitable for permissioned blockchains, not for permissionless blockchains. For the latters, both the throughput of transactions and computation burden are quite limited. The main computation on-chain of PoD-AS is computing with $O(n)$ hashes. The gas consumption would be unacceptable if the dataset is large. The experimental results show that on Ethereum, one transcation can only deliver data not greater than 350KB when the gas consumption almost reaches the block gas limit, the maxmium gas consumption of one single valid block. For permissioned blockchains, where the computation resources of chain nodes are rich, larger data can be delivered. Moreover, permissioned blockchains with high TPS can deliver data that can be cut into many pieces and each transaction delivers only one piece.

In this section, we present a variant protocol PoD-AS* aiming at reducing the computation on-chain from $O(n \cdot s)$ to $O(1)$, so as to support popular permissonless blockchains, particularly Ethereum. It uses zkSNARKs [19, 21, 29] to shift the majority of hash computation off-chain. The computation left on-chain is extremely lightweight, using only one hash computation to do public verification. However, as a tradeoff, the computation of $\mathcal{S}$ and $\mathcal{R}$ increases, lowering the throughput of data delivery. Let's review the computation on the blockchain, which mainly is key derivation from a seed, k_ω by using a random oracle. The algorithm of key derivation is $O(n \cdot s)$. By using zkSNARKs, the burden of hashes (key deriving) and key verification can be shifted from the blockchain to the sender $\mathcal{S}$, who constructs proofs $\{\pi\}_l$ and sends them to $\mathcal{R}$ showing that:

(1) the sender $\mathcal{S}$ has a knowledge of the key seed, k_ω, which can be derived to generate many keys;
(2) the hash value of the key seed is equal to h_ω;
(3) the keys derived from k_ω are precisely used for encrypting the data in the deliver-phase of PoD.

Thus in the deliver-phase, the receiver $\mathcal{R}$ can be convinced that the seed of keys can be hashed to h_ω. In the reveal phase, the confirmation submitted by the receiver is no more than a hash value, h_ω. The computation of the blockchain is thus reduced to a hash verification, which is $O(1)$. The protocol PoD-AS* is defined in Fig. 5.

Init-phase: In the init-phase, the scheme of zkSNARK needs to be setup in a trusted way. We omit the issue of trusted-setup here for simplicity. zkSNARK requires a prespecified circuit which is restricted to computing fixed-number of hashes. The setup generates a pair of keys, (ek, vk). ek is for $\mathcal{S}$ to generate proofs, and vk is for $\mathcal{R}$ to verify proofs. We introduce a parameter q specifying the number of hashes that the circuit can compute in one cycle. To support computing arbitrary hashes, we divide the data into many groups, each of which has at most n rows and each row has s slices. Thus the data size is $q \cdot n \cdot s$. The initializer also adds one column of slices for padding, m_{i0}.

$$\begin{bmatrix}
l=1 & \begin{array}{|c|ccccc} m_{1,1,0} & m_{1,1,1}, & m_{1,1,2}, & \ldots, & m_{1,1,s} \\ m_{1,2,0} & m_{1,2,1}, & m_{1,2,2}, & \ldots, & m_{1,2,s} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ m_{1,n,0} & m_{1,n,1}, & m_{1,n,2}, & \ldots, & m_{1,n,s} \end{array} \\
l=2 & \begin{array}{|c|ccccc} m_{2,1,0} & m_{2,1,1}, & m_{2,1,2}, & \ldots, & m_{2,1,s} \\ m_{2,2,0} & m_{2,2,1}, & m_{2,2,2}, & \ldots, & m_{2,2,s} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ m_{2,n,0} & m_{2,n,1}, & m_{2,n,2}, & \ldots, & m_{2,n,s} \end{array} \\
 & \cdots \\
l=q & \begin{array}{|c|ccccc} m_{q,1,0} & m_{q,1,1}, & m_{q,1,2}, & \ldots, & m_{q,1,s} \\ m_{q,2,0} & m_{q,2,1}, & m_{q,2,2}, & \ldots, & m_{q,2,s} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ m_{q,n,0} & m_{q,n,1}, & m_{q,n,2}, & \ldots, & m_{q,n,s} \end{array}
\end{bmatrix}^{(q \times n \times (s+1))}$$

We use notation $\mathbf{m}_{[1,q][1,n][0,s]}$ to denote the whole data file. The initializer generates $s+1$ group elements randomly.

$$\mathbf{u}_{[0,s]} = (u_0, u_1, u_2, \ldots, u_s), \text{ where } u_j \overset{\$}{\leftarrow} \mathbb{G}, j \in [0,s].$$

The initializer also computes $(n \cdot q)$ authenticators:

$$\sigma_{li} = \mathsf{Com}(m_{li1}, \ldots, m_{lis}; m_{li0}) = u_0^{m_{li0}} \cdot \prod_{j=1}^{s} u_j^{m_{lij}}, l \in [1, q], i \in [1, n].$$

The matrix of $\sigma_{[1,q][1,n]}$ is the public input and shared between the sender and the receiver. The size of the matrix is also $1/s$ of the data file. One additional job of the initializer is to setup the zkSNARK with a circuit, C, which computes hashes:

$$(ek, vk) = \mathsf{zkSetup}(C, 1^{\lambda}),$$

where ek is the evaluating key for the sender, vk is the verifying key for the receiver.

Deliver-phase: $\mathcal{S}$ firstly chooses two random numbers, a seed for generating keys k_{ω} and a blind factor k'_{ω}. Then $\mathcal{S}$ generates keys by a random oracle, $H(\cdot)$ with the seed k_{ω}:

$$k_{lij} = H(k_{\omega}, l, i, j); l \in [1, q], i \in [0, n], j \in [0, s].$$

For each group of rows, $\mathcal{S}$ generates one row of keys, $k_{l,0,[0,s]}$, not for encryption. These keys are used to blind the keys in the same columns by computing z_{lj}. For each row of keys, $\mathcal{S}$ generates a vector commitment, K_{li} for the i-th row in the l-th group:

$$K_{li} = \prod_{j=0}^{s} u_j^{k_{lij}}; l \in [1, q], i \in [0, n].$$

As the first step of deliver-phase, $\mathcal{S}$ sends all commitments, $\mathbf{K}_{[1,q][0,n]}$, to $\mathcal{R}$.

$$\begin{bmatrix} \boxed{k_{l,0,0} \; k_{l,0,1}, \; k_{l,0,2}, \; \ldots, \; k_{l,0,s}} & K_{l0} \\ k_{l,1,0} \; k_{l,1,1} \; k_{l,1,2} \; \cdots \; k_{l,1,s} & K_{l1} \\ k_{l,2,0} \; k_{l,2,1} \; k_{l,2,2} \; \cdots \; k_{l,2,s} & K_{l2} \\ \cdots \; \cdots \; \cdots \; \cdots & \cdots \\ k_{l,n,0} \; k_{l,n,1} \; k_{l,n,2} \; \cdots \; k_{l,n,s} & K_{ln} \end{bmatrix}.$$

$\mathcal{R}$ replies back a challenge number c in the second step. Then $\mathcal{S}$ encrypts the data with keys and the challenge number:

$$\overline{m}_{lij} = k_{lij} + m_{lij} \cdot c^i; l \in [1, q]; i \in [1, n], j \in [0, s]$$

For each group, $\mathcal{S}$ computes z_{lj} as follows:

$$z_{lj} = \sum_{i=0}^{n} k_{lij} \cdot c^i; l \in [1, q]; j \in [0, s].$$

We can see that each z_{lj} is the encoding of keys in one column.

$$\begin{array}{ccccc} \boxed{\begin{array}{c} k_{l,0,0} \\ k_{l,1,0} \\ k_{l,2,0} \\ \cdots \\ k_{l,n,0} \end{array}} & \begin{array}{c} k_{l,0,1}, \\ k_{l,1,1}, \\ k_{l,2,1}, \\ \cdots \\ k_{l,n,1}, \end{array} & \begin{array}{c} k_{l,0,2}, \\ k_{l,1,2}, \\ k_{l,2,2}, \\ \cdots \\ k_{l,n,2}, \end{array} & \begin{array}{c} \ldots, \\ \ldots, \\ \ldots, \\ \\ \ldots, \end{array} & \begin{array}{c} k_{l,0,s} \\ k_{l,1,s} \\ k_{l,2,s} \\ \cdots \\ k_{l,n,s} \end{array} \\ \hline z_{l,0} & z_{l,1} & z_{l,2} & \ldots, & z_{l,s} \end{array}.$$

They are used to verify the commitments $K_{[1,q][0,n]}$. Besides, $\mathcal{S}$ needs to generate zkSNARK proofs showing that:

$$C(l, j, c, z_{lj}, h_{\omega}; k_{\omega}, k'_{\omega}) = \begin{cases} (i) & k_{lij} = H(k_{\omega}, l, i, j); i \in [0, n] \\ (ii) & z_{lj} \overset{?}{=} \sum_{i=0}^{n} k_{lij} \cdot c^i \\ (iii) & h_{\omega} \overset{?}{=} H(k_{\omega} \parallel k'_{\omega}) \\ output : 1 & \text{if } (ii) \text{ and } (iii) \text{ hold} \\ output : 0 & \text{otherwise} \end{cases}$$

where l, j, c, z_{lj}, and h_{ω} are public inputs, and k_{ω}, k'_{ω} are private inputs. The circuit C outputs 1 if the equations (i) (ii) (iii) hold. Otherwise, it outputs 0. The proofs can be generated by using the evaluating key, common reference string and inputs:

$$\pi_{lj} \leftarrow \mathsf{zkProve}_{ek}((l, j, c, z_{lj}, h_{\omega}), 1, (k_{\omega}, k'_{\omega}))$$

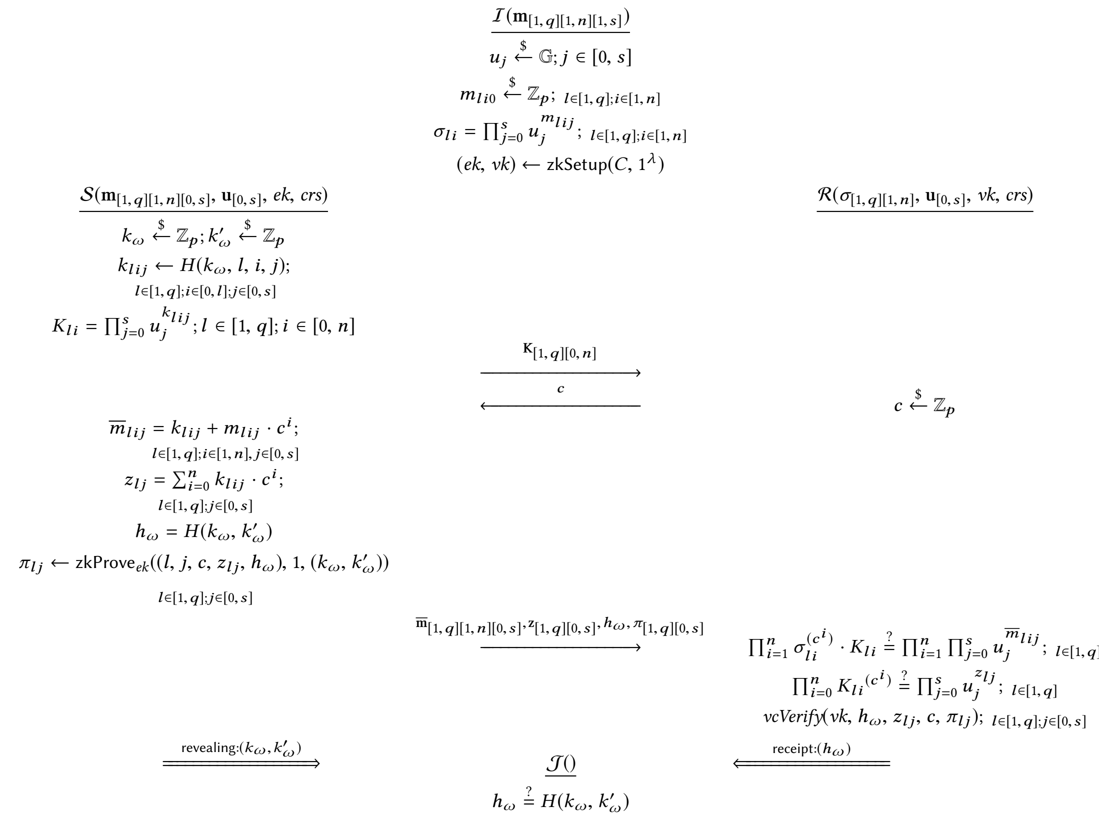

$\mathcal{I}(\mathbf{m}_{[1,q][1,n][1,s]})$
$u_j \xleftarrow{\$} \mathbb{G}; j \in [0, s]$
$m_{li0} \xleftarrow{\$} \mathbb{Z}_p;\ _{l\in[1,q];i\in[1,n]}$
$\sigma_{li} = \prod_{j=0}^{s} u_j^{m_{lij}};\ _{l\in[1,q];i\in[1,n]}$
$(ek, vk) \leftarrow \mathsf{zkSetup}(C, 1^\lambda)$
$\mathcal{S}(\mathbf{m}_{[1,q][1,n][0,s]}, \mathbf{u}_{[0,s]}, ek, crs)$
$k_\omega \xleftarrow{\$} \mathbb{Z}_p; k'_\omega \xleftarrow{\$} \mathbb{Z}_p$
$k_{lij} \leftarrow H(k_\omega, l, i, j);$
$_{l\in[1,q];i\in[0,l];j\in[0,s]}$
$K_{li} = \prod_{j=0}^{s} u_j^{k_{lij}}; l \in [1, q]; i \in [0, n]$
$\mathcal{R}(\sigma_{[1,q][1,n]}, \mathbf{u}_{[0,s]}, vk, crs)$
$\xrightarrow{\mathsf{K}_{[1,q][0,n]}}$
$\xleftarrow{c}$
$c \xleftarrow{\$} \mathbb{Z}_p$
$\overline{m}_{lij} = k_{lij} + m_{lij} \cdot c^i;$
$_{l\in[1,q];i\in[1,n],j\in[0,s]}$
$z_{lj} = \sum_{i=0}^{n} k_{lij} \cdot c^i;$
$_{l\in[1,q];j\in[0,s]}$
$h_\omega = H(k_\omega, k'_\omega)$
$\pi_{lj} \leftarrow \mathsf{zkProve}_{ek}((l, j, c, z_{lj}, h_\omega), 1, (k_\omega, k'_\omega))$
$_{l\in[1,q];j\in[0,s]}$
$\xrightarrow{\overline{\mathbf{m}}_{[1,q][1,n][0,s]}, \mathbf{z}_{[1,q][0,s]}, h_\omega, \pi_{[1,q][0,s]}}$
$\prod_{i=1}^{n} \sigma_{li}^{(c^i)} \cdot K_{li} \overset{?}{=} \prod_{i=1}^{n} \prod_{j=0}^{s} u_j^{\overline{m}_{lij}};\ _{l\in[1,q]}$
$\prod_{i=0}^{n} K_{li}^{(c^i)} \overset{?}{=} \prod_{j=0}^{s} u_j^{z_{lj}};\ _{l\in[1,q]}$
$vcVerify(vk, h_\omega, z_{lj}, c, \pi_{lj});\ _{l\in[1,q];j\in[0,s]}$
$\xRightarrow{\text{revealing}:(k_\omega, k'_\omega)}$
$\mathcal{J}()$
$h_\omega \overset{?}{=} H(k_\omega, k'_\omega)$
$\xLeftarrow{\text{receipt}:(h_\omega)}$

Fig. 5. PoD-AS*

In the third step, $\mathcal{S}$ sends four groups of values to $\mathcal{R}$:

(1) encrypted data, $\overline{\mathbf{m}}_{[1,q][1,n][0,s]}$,
(2) encoding of keys, $\mathbf{z}_{[1,q][0,s]}$,
(3) hash of seed, h_ω,
(4) vc-proofs, $\pi_{[1,q][0,s]}$.

At last step of deliver-phase, $\mathcal{R}$ firstly verifies if the encrypted data is correct w.r.t authenticators with the keys bound to the key-commitments:

$$\prod_{i=1}^{n} \sigma_{li}^{(c^i)} \cdot K_{li} \overset{?}{=} \prod_{i=1}^{n} \prod_{j=0}^{s} u_j^{\overline{m}_{lij}}; l \in [1, q].$$

Then $\mathcal{R}$ verifies the proof of knowledge of keys, w.r.t. the key-commitments:

$$\prod_{i=0}^{n} K_{li}^{(c^i)} \overset{?}{=} \prod_{j=0}^{s} u_j^{z_{lj}}; l \in [1, q].$$

Finally, $\mathcal{R}$ verifies the proofs showing that each column of keys were precisely derived from the pre-image of a hash value, h_ω:

$$\mathsf{zkVerify}_{vk}((l, j, c, z_{lj}, h_\omega), 1, \pi_{lj}); l \in [1, q]; j \in [0, s].$$

Reveal-phase: In the reveal phase, $\mathcal{R}$ sends a delivery-receipt (h_ω) to $\mathcal{J}$ if $\mathcal{R}$ accepts the data. Next $\mathcal{S}$ reveals the key (k_ω, k'_ω) to $\mathcal{J}$ if she checks that the receipt is correct. The job left to $\mathcal{J}$ is to verify if the key revealed is correct:

$$h_\omega \overset{?}{=} H(k_\omega, k'_\omega).$$

The protocol keeps provably *special HVZK* and *special knowledge soundness.*

Efficiency. The communication complexity of PoD-AS is $(q \cdot n \cdot s + 2q \cdot s + 2q \cdot n + 2)$, consisting of encrypted data $(q \cdot n \cdot s)$, the commitments of keys $(q \cdot n + 1)$, the encodings of keys $(q \cdot s)$, the proofs of keys $(q \cdot s)$ and authenticators $(q \cdot n)$. The computation of initialization is $(q \cdot n \cdot s)$ exponentiations with multiplication on $\mathbb{G}$, or (n) multi-exps. The computation of proving are roughly $(q \cdot n \cdot s)$ exponentiations with multiplication on $\mathbb{G}$, $q \cdot s$ zkSNARK proving, $(q \cdot n \cdot s)$ hashes (using a hash primitive to realize the random oracle function). The computation of verifying are mainly $(q \cdot n \cdot s + 2q \cdot n + q \cdot s)$ exponentiations on $\mathbb{G}$ and $(q \cdot s)$ zkSNARK verification. The PoD-AS* protocol has only 1 hash verification in the reveal-phase, extremely efficient for public verification.

3.5 PoD-CR: PoD in claim-or-refund mode

In this section, we present a variant of PoD protocol, PoD-CR, that is inspired by Fairswap. This protocol doesn't support atomic-swap, instead, it achieves fair trading by adopting the idea of claim-or-refund [6, 15]. The main advantage of the protocol is that the computation complexity off-chain is much smaller than PoD-AS*, and the computation on-chain is much smaller than PoD-AS. It is well-suited for delivering big data without requirment of immediate payment.

In PoD-CR, after receiving the revealings from $\mathcal{S}$, $\mathcal{J}$ doesn't verify the correctness of the key immediately. Instead, she leaves the work of verifying the revealings to $\mathcal{R}$. If $\mathcal{R}$ checks that the key is wrong, he is required to submit a proof of misbehavior to $\mathcal{J}$ showing that $\mathcal{S}$ is cheating. It is sufficient that $\mathcal{R}$ submits a proof only showing one single data slice/key is wrong, not about all data. Thus $\mathcal{J}$ only has to derive one single key to verify the proof in $O(1)$. In additional, $\mathcal{R}$ needs to provide a Merkle proof showing that the wrong key is a leaf node in the Merkle tree which is committed by $\mathcal{R}$ previously. Verifying the Merkle proofs requires the computation of $O(log(n))$. Therefore, the computation complexity of checking the entire proof of misbehavior is $O(log(n))$, compared to $O(n \cdot s)$ in the PoD-AS or PoD-AS* (atomic-swap mode).

As shown in Fig. **??**, there are three phases in PoD-CR. In the init-phase, the initializer computes authenticators in the same way as it does in the atomic-swap mode. The protocol has similar moves compared with atomic-swap mode.

Init-phase: In the init-phase, the data file is splitted into a block matrix of $n \times s$, with one additional column of random slices, m_{0i}, as it is in PoD-AS.

$$\left[\begin{array}{c|ccccc} m_{01} & m_{11}, & m_{12}, & \dots, & m_{1s} \\ m_{02} & m_{21}, & m_{22}, & \dots, & m_{2s} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ m_{0n} & m_{n1}, & m_{n2}, & \dots, & m_{ns} \end{array}\right]$$

The initializer needs to generate $n + 1$ group elements randomly, $\mathbf{u}_{[0,s]}$. Then she computes n authenticators, $\sigma_{[1,n]}$, which is shared by $\mathcal{S}$ and $\mathcal{R}$.

Delive-phase: The one-time keys generated by $\mathcal{S}$ used for encryption are also generated by the random oracle $H(\cdot)$. In the first move of the deliver-phase, $\mathcal{S}$ sends commitments of the derived keys:

$$K_{ij} = u_j^{k_{ij}}.$$

When $\mathcal{R}$ receives the commitments $\mathbf{K}_{[1,n][0,s]}$, he computes the Merkle tree root of them, and gets γ_k. A challenge number is randomly chosen from $\mathbb{Z}_p$ and sent to $\mathcal{S}$. The sender $\mathcal{S}$ encrypts m_{ij} and gets $\overline{m}_{ij}$, which are sent to $\mathcal{R}$.

$$\overline{m}_{ij} = k_{ij} + m_{ij} \cdot c^i; i \in [1, n], j \in [0, s]$$

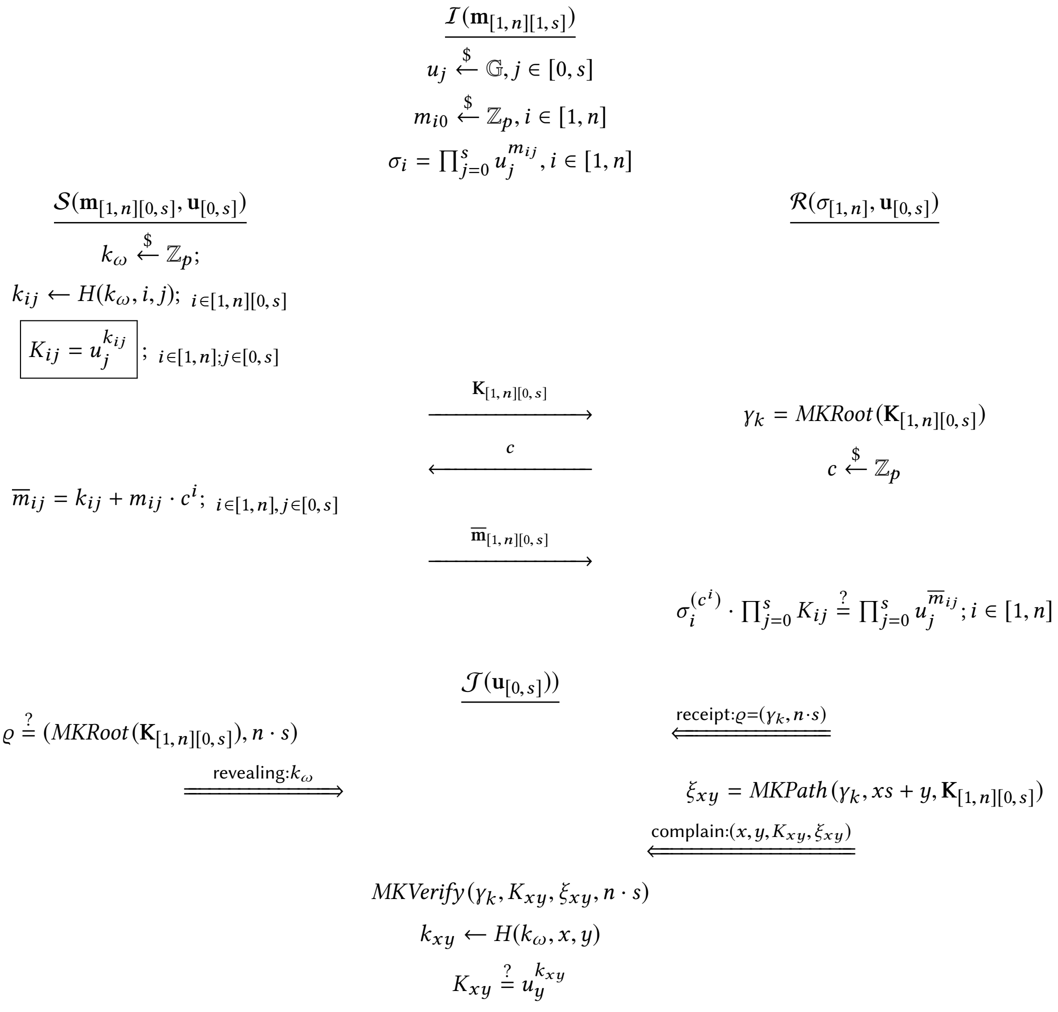

$\mathcal{I}(\mathbf{m}_{[1,n][1,s]})$
$u_j \xleftarrow{\$} \mathbb{G}, j \in [0,s]$
$m_{i0} \xleftarrow{\$} \mathbb{Z}_p, i \in [1,n]$
$\sigma_i = \prod_{j=0}^{s} u_j^{m_{ij}}, i \in [1,n]$
$\mathcal{S}(\mathbf{m}_{[1,n][0,s]}, \mathbf{u}_{[0,s]})$
$k_\omega \xleftarrow{\$} \mathbb{Z}_p;$
$k_{ij} \leftarrow H(k_\omega, i, j);\ i\in[1,n][0,s]$
$\boxed{K_{ij} = u_j^{k_{ij}}};\ i\in[1,n]; j\in[0,s]$
$\mathcal{R}(\sigma_{[1,n]}, \mathbf{u}_{[0,s]})$
$\mathbf{K}_{[1,n][0,s]}$
$\gamma_k = MKRoot(\mathbf{K}_{[1,n][0,s]})$
c
$c \xleftarrow{\$} \mathbb{Z}_p$
$\overline{m}_{ij} = k_{ij} + m_{ij} \cdot c^i;\ i\in[1,n], j\in[0,s]$
$\overline{\mathbf{m}}_{[1,n][0,s]}$
$\sigma_i^{(c^i)} \cdot \prod_{j=0}^{s} K_{ij} \overset{?}{=} \prod_{j=0}^{s} u_j^{\overline{m}_{ij}}; i \in [1,n]$
$\mathcal{J}(\mathbf{u}_{[0,s]}))$
receipt:$\varrho=(\gamma_k, n\cdot s)$
$\varrho \overset{?}{=} (MKRoot(\mathbf{K}_{[1,n][0,s]}), n \cdot s)$
revealing:k_ω
$\xi_{xy} = MKPath(\gamma_k, xs+y, \mathbf{K}_{[1,n][0,s]})$
complain:(x, y, K_{xy}, ξ_{xy})
$MKVerify(\gamma_k, K_{xy}, \xi_{xy}, n \cdot s)$
$k_{xy} \leftarrow H(k_\omega, x, y)$
$K_{xy} \overset{?}{=} u_y^{k_{xy}}$

Fig. 6. PoD-CR: Claim-or-refund mode

The receiver $\mathcal{R}$ verifies if the encrypted data are correct w.r.t. the authenticators and commitments of keys: for every i,

$$\sigma_i^{(c^i)} \cdot \prod_{j=0}^{s} K_{ij} \overset{?}{=} \prod_{j=0}^{s} u_j^{\overline{m}_{ij}}.$$

Reveal-phase: If $\mathcal{R}$ confirms the data in the deliver-phase, he submits $\mathcal{J}$ a delivery-receipt, $(\gamma_k, n\cdot s)$, which consists of slice numbers and the Merkle root of all of the commitments sent in the first-move of deliver-phase. Since the delivery-receipt is visible to $\mathcal{S}$, who can immediately verify if the Merkle root is correct. If it is, $\mathcal{S}$ reveals the seed k_ω to $\mathcal{J}$. Next, $\mathcal{R}$ derives all of the keys from k_ω, and verifies if every key k_{ij} is the opening to the commitment K_{ij}. Suppose for some (x, y), the key doesn't match its commitment:

$$K_{xy} \neq u_y^{k_{xy}}.$$

Then $\mathcal{R}$ needs to submit a proof of misbehavior, (x, y, K_{xy}, ξ_{xy}), where ξ_{xy} is the Merkle path of K_{xy}. It is computed as follows:

$$\xi_{xy} = MKPath(\gamma_k, x, y, \mathbf{K}_{[1,n][0,s]}).$$

The verification of $\mathcal{J}$ consists of two steps: (i) verify if K_{xy} is one of the leaves of the Merkle tree specified by γ_k:

$$MKVerify(\gamma_k, K_{xy}, \xi_{xy}, n \cdot s) \stackrel{?}{=} 1;$$

(ii) verify if the key k_{xy} is the opening to K_{xy}:

$$K_{xy} \stackrel{?}{=} u_y^{k_{xy}}.$$

Efficiency. The communication of PoD-CR is $(2ns + n + log(ns))$, consisting of encrypted data $(n \cdot s)$, the commitments of keys $(n \cdot s)$, the Merkle proof $log(n \cdot s)$, and authenticators n. The computation of initialization is $(n \cdot s)$ exponentiations with multiplication on $\mathbb{G}$, or (n) multi-exps. The computation of proving are roughly $(n \cdot s)$ exponentiations on $\mathbb{G}$, $(n \cdot s)$ hashes (using a hash primitive to realize the random oracle function), and $(n \cdot s)$ multiplications on $\mathbb{Z}_p$. The computation of verifying on delivery are mainly $(n \cdot s + 2n + s)$ exponentiations on $\mathbb{G}$; while the verification on reveal is $(n \cdot s)$ The computation of judge is $log(n \cdot s) + 1$ hash and 1 exponentiation on $\mathbb{G}$.

4 EXTENSION: QUERIES BY KEYWORDS

The PoD protocol is flexible enough to extend with other features so as to be used in more applications. The flexibility comes from the organization of the data. To deliver one data block is almost the same with to deliver big chunks of data in batches. Therefore, online data trading via PoD can also be seen as a trustworthy database, where anyone can obtain any block of the whole dataset by sending an index set to the seller. The indices, however, are unfriendly to database users who don't know the index of the block in which they are interested. Can PoD allows users to query as they do to a conventional database? The answer is Yes: PoD can support queries by keywords. In this section, we show how to extend PoD protocols to support queries by keywords. The queries are about membership and non-membership, such that a buyer may learn if data records with one keyword does exist. The results of queries are delivered in the same way with raw data in PoD protocols. The exchange should be fair and secure by using a blockchain-based trustless third party.

As aforementioned, a data file was organized into an array of $n \cdot s$ to reduce the size of authenticators σ. One of the advantages of array-structure is that it can be seen a *table* with multiple keys and values. Each row of the table is a multiple-key record, where some columns can be served as keywords. By the ECC curve underly we choose, the maximium size of a keyword column is 31-byte long, which will sufficient for encoding common datatype like name, ID, integer, enumerates and short arrays. In the first step, we show how to extend with membership and non- membership queries. The extended protocol allows Bob to send a keyword to Alice, and Alice may reply with all indices of blocks hit by the keyword. The extended protocol is still fair for both Alice and Bob. Notably, Alice cannot cheat Bob by replying with wrong indices, and Bob must pay for the query before learning the indices.

We use *Verifiable Random Function* (VRF) [8] schemes to build efficient proofs of membership and non-membership. In the init-phase, $\mathcal{I}$ specifies which column is used for queries, and generates an additional column of data used for query proofs. The basic idea of PoD-QKW is that the additional data is random numbers generated from the keywords column by using a private key, and they can be verified by $\mathcal{R}$ with a public key.

4.1 Verifiable random function

We first introduce a paring-based VRF scheme. Let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be a computable bilinear map with group $\mathbb{G}_1$'s support being $\mathbb{Z}_p$. Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be the generators of $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively.

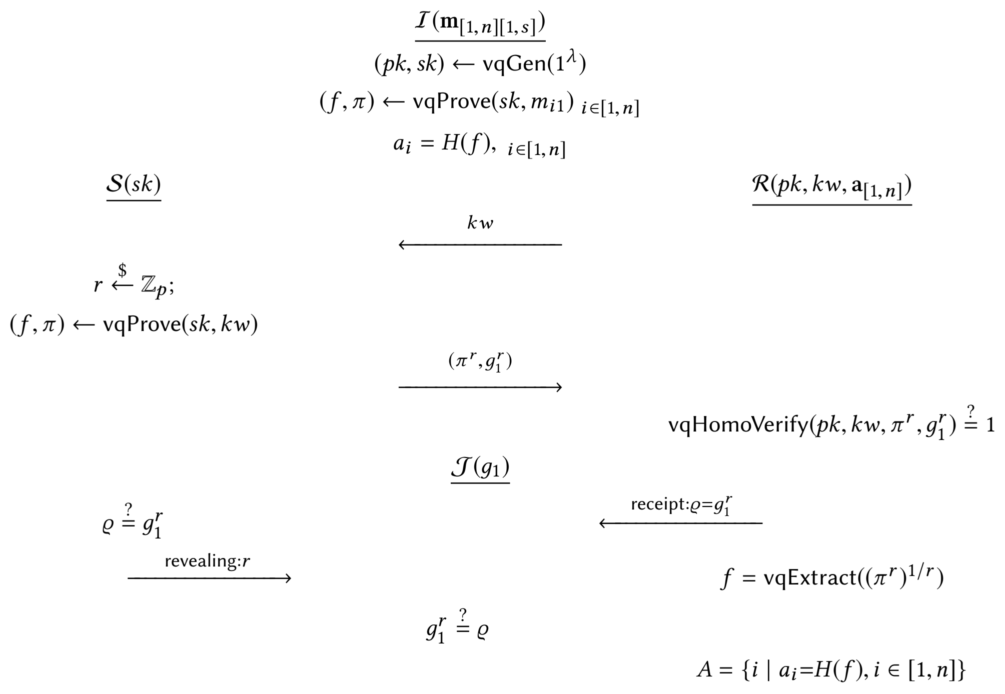

$\mathcal{I}(\mathbf{m}_{[1,n][1,s]})$
$(pk, sk) \leftarrow \mathsf{vqGen}(1^\lambda)$
$(f, \pi) \leftarrow \mathsf{vqProve}(sk, m_{i1})\ _{i\in[1,n]}$
$a_i = H(f),\ _{i\in[1,n]}$
$\mathcal{S}(sk)$
$\mathcal{R}(pk, kw, \mathrm{a}_{[1,n]})$
kw
$r \xleftarrow{\$} \mathbb{Z}_p;$
$(f, \pi) \leftarrow \mathsf{vqProve}(sk, kw)$
(π^r, g_1^r)
$\mathsf{vqHomoVerify}(pk, kw, \pi^r, g_1^r) \overset{?}{=} 1$
$\mathcal{J}(g_1)$
receipt:$\varrho = g_1^r$
$\varrho \overset{?}{=} g_1^r$
revealing:r
$f = \mathsf{vqExtract}((\pi^r)^{1/r})$
$g_1^r \overset{?}{=} \varrho$
$A = \{i \mid a_i = H(f), i \in [1,n]\}$

Fig. 7. PoD-QKW: Proof of Delivery of Queries with Keywords

Definition 4.1 (Verifiable random function). A VRF is an efficiently computable function $F : K \times X \to Y$ with four algorithms:

- $\mathsf{VRFGen}(1^\lambda)$ outputs a pair of keys (pk, sk) for some security parameter λ.
- $\mathsf{VRFProve}(sk, x)$ computes $F(sk, x), \pi(sk, x)$, where $\pi(sk, x)$ is a proof of correctness.
- $\mathsf{VRFVerify}(pk, x, y, \pi)$ verifies that $y = F(sk, x)$ using the proof π.
- $\mathsf{VRFHomoVerifyProof}(pk, x, \pi^r, g_1^r)$ verifies that $y = F(sk, x)$ using the one-time-pad encrypted proof π^r along with g_1^r.

4.2 PoD-QKW

The protocol is shown in Fig. 7.

Init-phase. In the init-phase, $\mathcal{I}$ organizes the data into an array of $n \cdot s$, each of which is a name-value record. The notation $kw_{[1,n]}$ specifies the set of all keywords.

$$name : kw_i, value : m_{i1}, m_{i2}, \ldots, m_{is}$$

The first column is for names (or keywords), and other columns are for values. $\mathcal{I}$ generates a pair of keys:

$$(pk, sk) \leftarrow \mathsf{VRFGen}(1^\lambda).$$

Then $\mathcal{I}$ computes one *random representation* for the name of each row:

$$(F(sk, kw_i), \pi(sk, kw_i)) \leftarrow \mathsf{VRFProve}(sk, kw_i).$$

$$a_i = H(F(sk, kw_i)), i \in [1, n].$$

At last, $\mathcal{I}$ prepends $a_{[1,n]}$ to the array:

$$\begin{array}{c|ccc} a_1, & m_{11}, & \ldots, & m_{1s} \\ a_2, & m_{21}, & \ldots, & m_{2s} \\ \ldots & \ldots & & \ldots \\ a_n, & m_{n1}, & \ldots, & m_{ns} \end{array}.$$

Deliver-phase. In this phase, $\mathcal{R}$ firstly sends a query with a keyword kw to $\mathcal{S}$, who picks up a random number, $r \in \mathbb{Z}_p$, and replies a membership/non-membership proof π^r encrypted by r, along with an element g_1^r. $\mathcal{S}$ then verifies the encrypted proof:

$$\mathsf{VRFHomoVerifyProof}(pk, kw, \pi^r, g_1^r) \stackrel{?}{=} 1.$$

If the proof is valid, $\mathcal{R}$ submits a delivery-receipt, g_1^r, to $\mathcal{J}$.

Reveal-phase. After $\mathcal{R}$ submits a g_1^r to $\mathcal{J}$ indicating she accepts the encrypted proof. Then $\mathcal{S}$ checks the receipt and reveal the random number r to $\mathcal{J}$, who finally verifies if the randomness matches the delivery-receipt.

$$g_1^r \stackrel{?}{=} \varrho.$$

Then $\mathcal{R}$ is able to decrypt the proof with the randomness revealed:

$$\pi = (\pi^r)^{1/r}.$$

From π, $\mathcal{R}$ can extract $F(sk, kw)$:

$$F(sk, kw) \longleftarrow \mathsf{vqExtract}(\pi).$$

Finally. $\mathcal{R}$ computes $H(F(sk, kw_i))$ and compares it with the vector $a_{[1,n]}$ to get all of the indices of the blocks with the same name kw.

4.3 BMR verifiable random function

The secure VRF we adopt is BMR-VRF [8] with large domain using nl − BDH assumption and the augmented cascade.

$\mathsf{Gen}(1^\lambda)$: Choose random generators $g_1 \in \mathbb{G}_1$, and $g_2, u \in \mathbb{G}_2$ and random values $s_1, s_2, \ldots, s_{32} \in \mathbb{Z}_p$; and output a pair of keys: pk and sk.

$$pk = (g_1, g_2, u, g_2^{s_1}, g_2^{s_2}, \ldots, g_2^{s_{32}})$$
$$sk = (g_1, g_2, u, s_1, s_2, \ldots, s_{32}).$$

$\mathsf{vqProve}(sk, kw)$: Define a function $F(sk, x)$ on input sk and $kw = kw[1], kw[2], \ldots, kw[32] \in [l]^n$:

$$F(sk, kw) = e(g_1^{[1/\prod_{k=1}^{32}(kw[j]_i + s_j)]}, u)$$

$$\pi(sk, kw) = (\pi_1, \pi_2, \ldots, \pi_{32}), \text{ where } \pi_i = g_1^{[1/\prod_{k=1}^{i}(kw[j]_i + s_j)]}.$$

The algorithm outputs $(F(sk, kw), \pi(sk, kw))$:

$$\mathsf{VRFProve}(sk, kw) = (F(sk, kw), \pi(sk, kw)).$$

$\mathsf{vqVerify}(pk, kw, y, \pi)$: The verification is an iteration over proof. First check:

$$e(\pi_i, g_2^{(kw[i])} g_2^{s_i}) \stackrel{?}{=} e(\pi_{i-1}, g_2), \text{ for } i \in [1, 32],$$

where $\pi_0 = g_1$. Then check: $e(\pi_{32}, u) \stackrel{?}{=} y$.

$\mathsf{vqHomoVerify}(pk, kw, \pi^r, g_1^r)$: The verification is an iteration over proof. First check:

$$e(\pi_i^r, g_2^{(kw[i])} g_2^{s_i}) \stackrel{?}{=} e(\pi_{i-1}^r, g_2), \text{ for } i \in [1, 32],$$

where $\pi_0^r = g_1^r$. The second check is ignored.

We may argue that the protocol cannot ensure that for one record, the value does match the keyword, even if the proof shows that the keyword is related to the index authenticator via VRF.

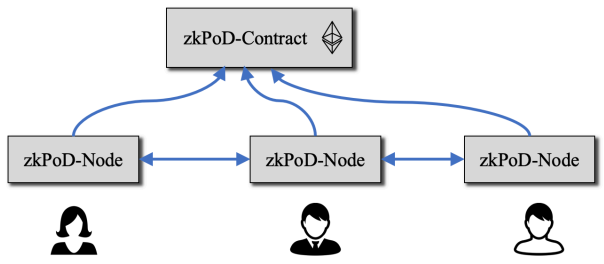


Fig. 8. zkPoD System

However, the protocol ensures that the record is bound with index-authenticators such that $\mathcal{S}$ cannot change. The protocol supports multiple keywords straightforwardly.

5 ZKPOD: A PRACTICAL SYSTEM FOR FAIR DATA EXCHANGING

In this section, we illustrate zkPoD, a pratical system for fair data exchanging. It is built on PoD protocols, supporting data exchanging for any two parties who don't have to trust each other. The system uses Ethereum as the blockchain to implement the role of $\mathcal{J}$. In zkPoD, there are mainly three parties, a buyer (Alice), a seller (Bob) and a judge (Julia). Julia, the role of $\mathcal{J}$, is implemented as smart contracts on Ethereum (any blockchain with smart contracts can be $\mathcal{J}$).

As shown in Fig. 8, the system consists of nodes and one smart contract deployed on Ethereum. One node can either be a buyer or a seller. Nodes can communicate with each other through various networking protocols, *e.g.,* TCP/IP, BitTorrent, IPFS, or SyncThing. Julia the contract is designed for multiple users. In other words, many buyers and sellers can share one smart contract. The zkPoD system deploys one single zkPoD-Contract on Ethereum when the system does setup. The zkPoD will support more blockchains in the future, *e.g.,* Bitcoin.

5.1 Making PoD practical

There are certainly many issues to be tackled when we realizing the PoD protocols to build a practical system. from the ways of price negotiation to avoiding the potential security risks on Ethereum, a permissionless blockchain where anyone can access to the on-chain data and send transactions.

Fariness. In theory, the PoD protocols are strong fair such that any party cannot gain more advantages over the other. However, it is sophisticated to measure "advantages" in the real world, where computation power, network bandwidth, interferences and DDos attacks are almost impossible to formalize. We introduce economic remedies to mitigate the "unfairness" occuring in the zkPoD practice. One is the way of deposits before Alice and Bob begin to interact. They are both required to deposite ethers into the contract and behave honestly. The deposite might be forfeited by the smart contract if any of them cheated.

Security. Ethereum, like all other blockchain systems, is complicated and subtle in security. There have been many security events occurred on Ethereum varying from consensus protocols to smart contracts. Two attack patterns may affect the security of zkPoD: (i) front-running attack and (ii) stuffing attack. When a user reveals a secret (*e.g.,* a key) to the chain, an adversary may get learn about the secret and insert a new transaction to reveal the secret with higher gas price as if the adversary is the one having the secret. The stuffing attack is more like a (Deny-of-Service) DOS attack, *i.e.,* an adversary may clog blocks by producing many elaborate transactions with high gas consumption in order to prevent miners from accept those others' transactions. The stuffing attack may thwart proofs of misbehaviors submitted by buyers if they are in the claim-or-refund protocols, *e.g.,* PoD-CR.

Gas consumption. Every interaction with smart contracts will consume gas, which is fee paid to miners. Gas system isn't only the vital part of the Ethereum ecosystem, but also an important issue of zkPoD. The main design of zkPoD is to reduce the gas consumption as possible as we can. One way of reducing gas cost is to decrease the number of interactions between nodes and the contract. The other is to reduce the computation burden of smart contracts.

Engineering performance. The data exchanged could be up to GBs, even TBs. The size of data will increase unstoppably as bandwidth improves. To support fair exchange of data with large size can be challenging, posing three main directions of algorithmic optimization, the computation on zero-knowledge proofs (proving and verifying), the amount of communication, and the computation on key-receipt verification on the chain. From the beginning, we've been upgrading both protocols and algorithms to the level where users can exchange data with each other using laptops at home. We also plan to optimize the protocols so as to support mobile devices.

5.2 System setup

In the phase of zkPoD system setup, there are three main jobs to be done:

(i) generating the random generators of the group $\mathbb{G}$;
(ii) generating the common reference strings of zkSNARK scheme;
(iii) deploying the smart contract on Ethereum.

Some important system parameters need to be decided in the phase.

NUMS Generators. Remember that in the init-phase of PoD protocols, the initializer needs to generate $\mathbf{u}_{[0,s]}$, $s+1$ elements in groups. The generators must be generated randomly, such that nobody knows the discrete logarithmic relation between any of u_j and u_k for any $j,k \in [0,s]$, and $j \neq k$. Otherwise, the binding property of Pedersen commitments wouldn't hold. For example, if Alice knows α such that $u_j = u_0^{\alpha}$, she may forge data. They are called "nothing-up-my-sleeve numbers" generated by a hash function, $H_2 : \{1,0\}^* \mapsto \mathbb{G}$.

$$u_j = H_2(\mathit{prefix} \parallel j); j \in [0,s].$$

By the hash function, the discrete logarithmic relations between these points are hard to compute.

Precomputing. The vast majority of computation costs are exponentiations over generators. One effective way to accelerate the computing is to precompute many exponents: $u^2, u^4, u^8, u^{32}, u^{1024}, \ldots$ in the phase of system setup. The precomputings are stored into a local file, which is loaded into memory when Alice or Bob starts to do transactions.

Trusted-setup of zkSNARK. The system of zkPoD realizes zkSNARKs by using libsnark [4], which needs a phase of trusted-setup to generate common reference strings, or a pair of *ek* and *vk*. In the trusted-setup, a few random numbers are chosen to compute *ek* and *vk*. But these random numbers cannot be known by both provers and verifiers. Otherwise, any party who knows of these numbers would break the security of the protocols. Thus the trusted-setup should be done by a trusted-third party, or through MPC protocols [9]. As shown in Sec. 3.4, the setup $\mathsf{zkSetup}(C, 1^{\lambda})$

needs a circuit as a parameter. In zkPoD, there is only one predefined arithmetic circuit computing hashes and the setup is done using a MPC protocol to ensure that the setup wouldn't introduce any security vulnerabilities.

Smart contract. All nodes of zkPoD interact with one globally single contract deployed on Ethereum. In the phase of system setup, there are a few parameters to be set when the contract is creating. The first is the value of complain-timeout that is a time windows within which Bob has to submit a proof of misbehavior of Alice if they exhange data by the protocol of PoD-CR. If Bob failed to submit the proof before the deadline, his coins would be transferred to Alice, even if Alice was cheating. The second parameter is the value of withdraw-timeout that Bob must be waiting before he can withdraw his deposits. There are some other parameters, about the ECC computation, to be explained in Sec. 6. Certainly, we can redeploy the smart contract if it is updated, and any zkPoD-node can connect to multiple contracts. But if two zkPoD-nodes exchange data, they have to connect to the same contract.

5.3 Alice and Bob: Node setup

For users, Alice or Bob, they have to setup the node before doing exchange transactions. The zkPoD system reuse the account system of Ethereum. In other words, every user has an ID, specified by a public and private key pair. The node can import a keystore file from any Ethereum wallet, or create new key pairs for users. Users may also give node a netID, which may be an IP-address, or an address of P2P networks. Currenly, zkPoD only supports IP-addresses to be netIDs. Alice may tell Bob her netID offline before their nodes establish the connection. Alice and Bob need to set the same address of Julia, the judge contract.

$$
\begin{array}{cccl}
(\textsc{PublicKey}) & \mathsf{pk} & ::= & \langle \textit{eth-address} \rangle \\
(\textsc{PrivateKey}) & \mathsf{sk} & ::= & \langle \textit{eth-key} \rangle \\
(\textsc{NetID}) & \mathsf{netID} & ::= & \langle \textit{ip:port} \rangle \mid \cdots
\end{array}
$$

5.4 Data publishing

Before exchanging data, Alice is required to initialize the data file and publish the meta information to Bob and Julia. A file F is composed of a name fname, a state fstat, a mode fmode, a protocol proto, s (slice numbers per row), N (total block numbers), the Merkle root of authenticators γ_σ, and a keyword table for querying ψ. The state of a data file can either be Published on the blockchain or Removed. Alice may choose *binary-mode* or *table-mode* to initialize a file according to the structure of the data file. In either mode, Alice sends the record of F to Julia the contract. Alice can assign a protocol to the file tag specfiying which protocol Bob should use.

$$
\begin{array}{cccl}
(\textsc{FileTag}) & \mathsf{F} & ::= & (\mathsf{fname}, \mathsf{fstat}, \mathsf{fmode}, \mathsf{proto}, s, N, \gamma_\sigma, \psi) \\
(\textsc{Name}) & \mathsf{fname} & ::= & \langle \textit{string} \rangle \\
(\textsc{FileState}) & \mathsf{fstat} & ::= & \mathsf{Published} \mid \mathsf{Removed} \\
(\textsc{Mode}) & \mathsf{fmode} & ::= & \mathsf{Binary} \mid \mathsf{Table}\ (j_1, j_2, \ldots, j_w) \\
(\textsc{Protocol}) & \mathsf{proto} & ::= & \mathsf{AS} \mid \mathsf{AS}^* \mid \mathsf{CR} \mid \mathsf{QKW}
\end{array}
$$

If the data file is initialized in the binary mode, the organization of the data file can be viewed as a matrix of $N \cdot s$, as it is in the PoD protocols. We assume the an initialized data file in binary mode has N row. Each row is called a block and there are s slices in each block. Please note that each slice is 31 bytes. In the initialization, each row is prepended a random slice for padding. There are vertical padding for alignment such that N is a mutiplication of L_{hash}. After the initialization, the

Merkle root of authenticators should be computed. Alice keeps all of the authenticators, and she can send all or parts of them to Bob on demand in the transaction.

The data file initialized in the table mode supports queries with keywords. In the initialization, Alice chooses the value of w, which should be less than s in the smart contract. In the table-mode, s is the number of columns. Please note that in the table mode, the size of a column should be less than 31 bytes. If in table mode, there are two more outputs: (iii) index-meta and (iv) the hash of index-meta. As we explained in Sec. 4, index-meta data is associated with the value of the indexed column.

5.5 Global smart contract

$$\begin{array}{ccll}
(\textsc{SellerState}) & A & ::= & (\mathsf{pk}_A, \mathsf{sk}_A, [\mathsf{f}_1, \mathsf{f}_2, \ldots, \mathsf{f}_n]) \\
(\textsc{BuyerState}) & B & ::= & (\mathsf{pk}_B, \mathsf{sk}_B) \\
(\textsc{JudgeState}) & J & ::= & (\mathsf{pk}_J, \mathbf{F}, \mathbf{V}, \mathbf{U}, \mathbf{D}) \\
(\textsc{FileSet}) & \mathbf{F} & ::= & \{\mathsf{f} \rightsquigarrow F\}^* \\
(\textsc{RevealSet}) & \mathbf{V} & ::= & \{\mathsf{sid} \rightsquigarrow (\mathsf{T}_{reveal}, \mathsf{V})\}^* \\
(\textsc{Revealing}) & \mathsf{V} & ::= & \mathsf{CR}\ (k_\omega) \mid \mathsf{AS}\ (k_\omega) \mid \mathsf{AS}^*\ (k_\omega, k'_\omega) \\
(\textsc{ReceiptSet}) & \mathbf{U} & ::= & \{\mathsf{sid} \rightsquigarrow \mathsf{U}\}^* \\
(\textsc{Receipt}) & \mathsf{U} & ::= & \mathsf{CR}\ (\mathsf{sid}, \mathsf{pk}_A, \mathsf{pk}_B, \mathsf{price}_{total}, \mathsf{T}_{receipt}, \gamma, n) \\
 & & \mid & \mathsf{AS}\ (\mathsf{sid}, \mathsf{pk}_A, \mathsf{pk}_B, \mathsf{price}_{total}, \mathsf{T}_{receipt}, c, z, n) \\
 & & \mid & \mathsf{AS}^*\ (\mathsf{sid}, \mathsf{pk}_A, \mathsf{pk}_B, \mathsf{price}_{total}, \mathsf{T}_{receipt}, h) \\
(\textsc{Deposits}) & \mathbf{D} & ::= & \{\mathsf{f} \rightsquigarrow (eth, \mathsf{dstat})\}^* \\
(\textsc{DepositState}) & \mathsf{dstat} & ::= & withdrawing \mid deposited
\end{array}$$

5.6 Deposit and withdraw

The most important thing between Alice and Bob is what is the price of data. The price is discussed off-line by Alice and Bob, and the price is dynamic according to the relation between supply and demand. Alice doesn't put the price into the contract since the modification of value in the contract is too expensive and slow. Here the basic steps of pricing in zkPoD:

(1) Alice and Bob reach an agreement on price.
(2) Bob writes the price into the delivery-receipt when he received the encrypted-data.
(3) Alice continues the protocol if she agrees on the price, or quits otherwise.
(4) The contract computes the ethers paid to Alice by the price in the receipt.

In zkPoD, Bob should deposit sufficient ethers into the contract before Alice reveals the key. Alice needs to check if the coins deposited by Bob can cover the payment.

Bob deposits ethers to the smart contract along with the address of Alice. If Bob wants to do transaction with Carol, Bob has to deposit again with Carol's (ETH) address. Bob may deposit more ethers into the contract at any time. If Alice finds that Bob's deposit is under payment, she can ping Bob to top up. Please note that the deposits to different sellers are separated, not shared. Bob may ask the contract to withdraw the ethers he deposited. The contract then sets a flag showing the ethers are in the "withdrawing" state. At this moment, Alice may be sending the delivery-receipt and the key to the contract. If so, the contract can cancel the state of "withdrawing". If after a time period of $\mathsf{T}_{withdraw}$, the state is still in "withdrawing", it will be changed into "deposited" automactically. Bob may withdraw the ethers immediately. Or, Bob can explicitly change state from "withdrawing" to "deposited" by depositing ethers to Alice (same seller) in the contract.

5.7 Security issues on Ethereum

We model the executions of zkPoD as follows:

$$
\begin{array}{rcll}
(\textsc{World}) & \mathsf{W} & ::= & (A, B, J, S) \\
(\textsc{Session}) & \mathsf{S} & ::= & (\mathsf{sid}, \mathsf{pk}_A, \mathsf{pk}_B, \mathsf{f}, \mathsf{fmode}, \mathsf{proto}, \mathit{step}) \\
(\textsc{EthTransaction}) & \mathsf{Tx} & ::= & (\mathsf{pk}, \mathsf{m}, \mathit{eth}) \\
(\textsc{Message}) & \mathsf{m} & ::= & A \rightarrow B(\cdots) \mid B \rightarrow A(\cdots) \\
 & & \mid & A \rightarrow J(\cdots) \mid B \rightarrow J(\cdots) \\
(\textsc{Ether}) & \mathit{eth} & ::= & \langle n \text{ ethers} \rangle
\end{array}
$$

We use W do specify the state of zkPoD system. It consists of four parts, the states of sellers **A**, the states of buyers **B**, the state of the judge J, the set of sessions S.

$$(W, M) \longmapsto W', \text{ if } A \rightarrow B \text{ or } A \leftarrow B$$

To formalize the asynchronism of blockchain transactions, we introduce two binary relations. One relation $(\mathsf{W}, \mathsf{m}) \Longrightarrow (\mathsf{W}'; \mathsf{Tx})$ specifies the step where a transction Tx is sent to the blockchain and W changes to W'. The other relation $(\mathsf{W}; [\mathsf{Tx}, \mathsf{Tx}']) \circlearrowleft (\mathsf{W}'; [\mathsf{Tx}'])$ specifies the step where the transaction Tx is accepted by the blockchain and W changes to W'.

$$(\mathsf{W}, \mathsf{m}; [\mathsf{Tx}]) \Longrightarrow (\mathsf{W}'; [\mathsf{Tx}] :: [\mathsf{Tx}'])$$
$$(\mathsf{W}; [\mathsf{Tx}_1] :: [\mathsf{Tx}] :: [\mathsf{Tx}_2]) \circlearrowleft (\mathsf{W}'; [\mathsf{Tx}_1] :: [\mathsf{Tx}_2])$$

Front-running attack. When a normal user sends a transaction calling a constract with sensitive parameters, the transaction is broadcasted to all miners. A malicious miner may learn the parameters inside the transaction before it is packed into a block. The miner, possibly, conduct a *front-running attack* by create a higher-priced transaction with the sensitive parameters pretending to be knowing the parameters. The forged transaction might be packed into blockchain ahead of the one sent from the normal user because miners generally choose those transactions with higher gas cost. When considering Ethereum security, one shall not assume that her transactions are packed before some others. Any smart contract shall not accept the transactions with secret inputs from unknow users/addresses. For some applications, users should send commitments (to the secrets) to the contract before they send the plaintext of secrets.

Stuffing attack. The average speed of block generation is around one block per 15 seconds. Each block consists of a list of transactions, the order of which is determined by miners. Generally, a miner chooses transactions with higher Gas consumption, since Gas consumption is the fee paid to the miner as the incentives of generating blocks. There is a system-wide parameter called *Gas-Limit* of the block that determines the number of transactions. The sum of the gas cost of all transactions cannot be higher than the Gas-Limit value. A malicious user of Ethereum thus can conduct *stuffing attacks* [**?**], discovered firstly in a well-known blockchain-game. The malicious user can initiate many transactions with switches which can turn the transactions into "active". At first, these inactive transactions are not interesting for miners, since the gas cost of these transactions are quite low when their switches are off. The miners then put them into their memory pools, waiting lists for transactions before being packed into blocks. After a while, these malicious transactions can be widespread over the memory pools of many miners. If seeing victim transactions, the attacker starts to attack by turning on the switches, raising the gas cost to a very high level. Thus miners much more likely pack these malicious transactions to build a new block rather than victim transactions sent by others. The attacker clogs the blockchain in a short period of time, and regular transactions will delay being packed. In the claim-or-refund protocols of zkPoD, the contract requires that Bob should submit the proof of misbehavior of Alice within a time period. If Alice was cheating and she might be motivated to conduct the stuffing attacks such that Bob's proofs cannot be seen by Julia the contract before the malicious transactions are all packed into blocks. If the gains through the attacks, the payment of Bob is higher than the cost of conducting the attacks, Alice will definitely have incentives to do the attacks. Therefore in the protocols of zkPoD, all of the parameters about time-windows should be long enough such that the cost of attacks would be unaffordable for Alice.

5.8 Interaction of data exchange

There are a few differences between interactions presented in this seciton and the protocols presented in Sec. 3. Aformentioned, we merge the transactions sent to the blockchain into one single transaction to save the trading time. We use one single judge in the zkPoD system, therefore introducing session-ID and account-ID (specified by the public keys of sellers and buyers). Regarding the asynchronism of communications of Ethereums, we introduces time limits to ensure the termination property of the interactions. We then explain the two protocols, zkPoD-AS* and zkPoD-CR step by step.

zkPoD-AS*. Suppose Alice published a data file and Bob begins to initiate the transaction of data exchange, they interact with each other with following the steps below:

(0) Alice and Bob negotiate the price of the data offline.
(1) Bob establishes the communication session with Alice and gets a session ID sid from Alice.
(2) Bob sends a request to Alice, (U), where U is the index-set of data blocks. $\mathsf{m}_{B\to A} = (\mathsf{sid}, U)$
(3) Alice sends encrypted data $\overline{m}$ and authenticators σ to Bob. $\mathsf{m}_{A\to B} = (\mathsf{sid}, \sigma, K, \overline{m}, z, k_\omega, \pi)$
(4) Bob verifies $\overline{m}$ and σ and replies back a delivery-receipt to Alice, $\mathsf{m}_{B\to A} = (\mathsf{U}, \mathsf{sig}_{receipt})$, $\mathsf{U}_B = (\mathsf{sid}, \mathsf{pk}_A, \mathsf{pk}_B, \mathsf{price}_{total}, k_\omega, \mathsf{T}_{receipt})$. The receipt should be signed by Bob's private key.
(5) Alice checks if the receipt is correct as well as Bob's signature. Alice also checks through the contract if Bob's deposit is greater than the price. Alice then checks if the time before due is safe ($\mathsf{T}_{receipt}$). If all of the condition are true, Alice finally submits both the seed of keys and the receipt signed by Bob to the contract. $\mathsf{m}_{A\to J} = (\mathsf{V}_A, \mathsf{U}_B, \mathsf{sig}_{receipt})$, where $\mathsf{V}_A = (k_\omega, k'_\omega)$.
(6) The contract first verifies Bob's signature and then verifies if the key-seed is corresponding to the receipt. If the two conditions are true, the contract transfers the ETHs to Alice, otherwise does nothing.

zkPoD-CR. The protocol is based on PoD-CR. We also assume that

(0) Alice and Bob negotiate the price of the data.
(1) Bob establishes the communication session with Alice and gets a session ID sid from Alice.
(2) Bob sends a request to Alice, (U), where U is the index-set of data blocks. $\mathsf{m}_{B\to A} = (\mathsf{sid}, U)$
(3) Alice sends encrypted data $\overline{m}$ and authenticators σ to Bob. $\mathsf{m}_{A\to B} = (\mathsf{sid}, \sigma, K, \overline{m})$
(4) Bob verifies $\overline{m}$ and σ and replies back a receipt to Alice, $\mathsf{m}_{B\to A} = (\mathsf{U}, \mathsf{sig}_{receipt})$, where $\mathsf{U}_B = (\mathsf{sid}, \mathsf{pk}_A, \mathsf{pk}_B, \mathsf{price}_{total}, n, \gamma_k, \mathsf{T}_{receipt})$. The receipt should be signed by Bob's private key.
(5) Alice checks if the receipt is correct as well as Bob's signature. Alice also checks through the contract if Bob's deposit is greater than the price. Alice then checks if the time before due is safe ($\mathsf{T}_{receipt}$). If all of the condition are true, Alice finally submits both the seed of keys and the receipt signed by Bob to the contract. $\mathsf{m}_{A\to J} = (\mathsf{V}_A, \mathsf{U}_B, \mathsf{sig}_{receipt})$, where $\mathsf{V}_A = k_\omega$.
(6) The contract stores the current time and waits for Bob's complain. After a period of time (T_c), Alice can withdraw ETHs specified by the same price in the receipt.

6 IMPLEMENTATION AND EVALUATION

6.1 Implementation

We implemented a prototype of zkPoD that is composed of three parts: zkPoD-lib, zkPoD-node, and zkPoD-contract. The zkPoD-lib is a library written in C++ with Golang bindings. It fully implements

Protocol	Init	Proving	Verification (on delivery)	Verification (on reveal)	Communication	Gas Cost (Ethereum)	Data/Tx (Ethereum)
zkPoD-CR	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(2N)$	$O(\log(N))$	< 100 TiB
zkPoD-AS	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	< 350 KiB
zkPoD-AS*	$O(N)$	$O(N) + O(N/q)_\pi$	$O(N) + O(N/q)_\pi$	$O(1)$	$O(N)$	$O(1)$	Unlimited

Fig. 9. Efficiency Overview

Protocol	Throughput	Prover (s)	Verifier (s)	Decrypt (s)	Communication (MiB)	Gas Cost
zkPoD-CR	3.39 MiB/s	124	119	82	2215	159,072
zkPoD-AS	3.91 MiB/s	130	131	4.187	2215	N/A
zkPoD-AS*	35 KiB/s	34540	344	498	2226	183,485

Fig. 10. PoD Benchmark. *(N=1024MiB, s=64, thread_num=12).*

PoD protocols, including zkPoD-AS, zkPoD-AS* and zkPoD-CR. It also provides a command line interface which is friendly for testing and debugging. zkPoD-lib has four main parts as follows:

- pod-setup: initializing system and generating public parameters.
- pod-publish: processing data and computing authenticators.
- pod-core: supporting interaction between two parties, implementing PoD protocols.
- pod-go: providing golang bindings.

The zkPoD-node is written in golang. It implements data exchanging, interacting with contracts, networking and key management. zkPoD-node also provides HTTP-APIs for building UI. It is now shipped with a command line interface. The zkPoD-contract implements deposits management, protocol verification (work of Julia). It is highly optimized for gas consumption.

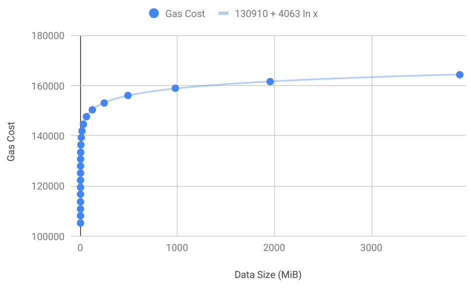
Currently, zkPoD system supports Linux (Ubuntu >16.04), Windows 10, and MacOSX (> 10.12). More tests are needed on more OSes. It requires 4GB RAM and 1GB storage at least.

6.2 Evaluation

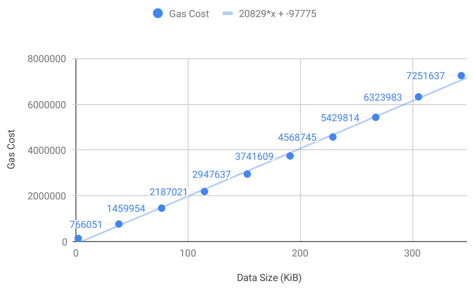
We experiment on an x86-64 server with Ubuntu 16.04 installed. The server has a 6-core CPU (Intel i7-8700K 3.70GHz) with 32GB of RAM.

Efficiency overview. As shown in Fig. 9, the protocol of zkPoD-AS is fastest with the speed of 3.91 MiB/s. However, the performance of zkPoD-AS is limited by a blockchain with high cost or low TPS. On Ethereum, a seller can only deliver 350KiB data per transaction. If users choose zkPoD-CR, the throughput of delivery is fast, 3.39 MiB/s on average. The cost of smart contract is rather low so as to support data up to 100 TiB per transaction. The protocol of zkPoD-AS* has lowest data throughput, only 35 KiB/s because generating zkSNARK proofs are very costly. But it has the best gas cost on Ethereum with a verification algorithm in $O(1)$. The protocol zkPoD-AS* in theory supports unlimited data per transaction.

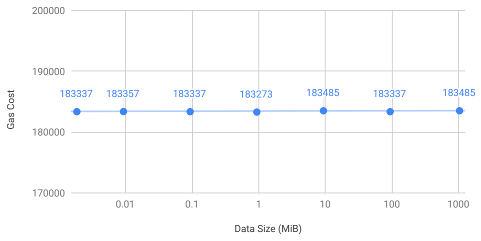
Suppose we have a 1GB data file to deliver, we compare the performance of the three protocols in the time overhead of proving-before-delivery, verification-on-delivery and decryption-after-revealing, as shown in Fig. 10. In zkPoD-AS*, the time overhead of proving-before-delivery is about 10 hours, which is the exact bottleneck of data throughput. Also, the overhead of decryption-after-revealing is high, about 8 minutes, since zkPoD-AS* uses circuit-friendly MIMC-Inv as the hash function for deriving one-time keys. While in zkPoD-AS, zkPoD uses SHA-3 as the hash function, which is much more efficient than MIMC-Inv. As for the gas cost, zkPoD-CR wins out, when the data size is less than 1GB.



(a) zkPoD-CR



(b) zkPoD-AS



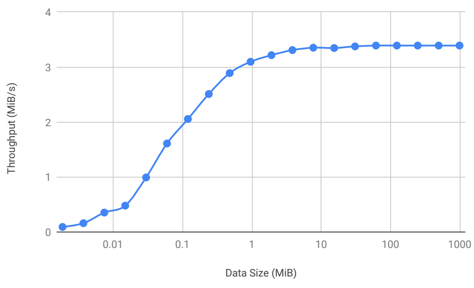
(c) zkPoD-AS*

Fig. 11. Gas cost on Ethereum

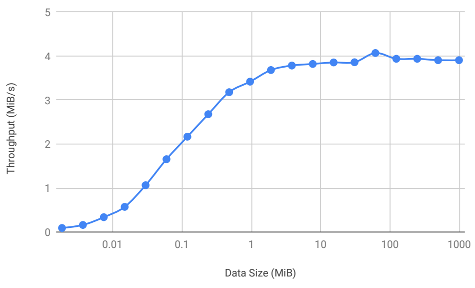
Gas consumption. In Fig. 11, we show the gas costs on Ethereum when data size varies. In zkPoD-CR, the gas consumption of verifying proofs of misbehavior has logarithmic correlation with requested file size due to large gas cost in Merkle proofs. The complexity of $O(\log(n))$ enables zkPoD-CR to support huge data files. The test data shows that 5-GiB-size file costs only 167,635 gas and the average gas cost increases 9,366 when the file size is ten times of the original size. In zkPoD-AS, a seller submits proof to the contract directly and the gas consumption is linearly correlated to the requested file size due to the contract gas cost of $n \cdot s$ loop. The graph shows that the gas cost is around 7,251,637 when the requested file size is 343.3 KiB, which is close to the ethereum block gas limit. While in zkPoD-AS*, the gas consumption is a constant around 183,500 no matter how much data is delivered.

Delivery capacity. In Fig. 12, we show the throughput of PoD core (including proving and verifying) for different data sizes. Here the number of slices in one block is set to $s = 64$. It shows the relation between pod-core processing speed and file sizes under two modes (n is exponential to 2). The first two graphs, (a) and (b), change in the similar way. As the file size grows, the average pod-core processing speed increases toward 3MiB and then becomes stable. The speed under zkPoD-AS protocol is generally higher than that under zkPoD-CR protocol. Please notice that the overhead of contract verification is not considered in this scene. In the graph of Fig. 12-(c), the average throughput approaches 30KiB. We can see some zigzags that are caused by padding blocks added for alignment. They dispear when the size is larger than 10MiB.

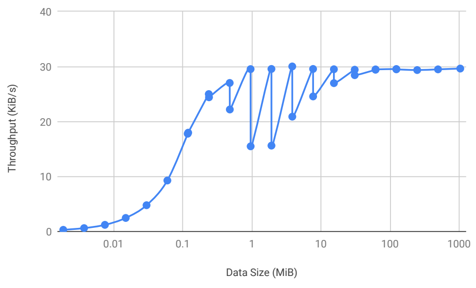
Computation of proving and verification. We measure the computation of PoD-core at different parts: (i) the proving computation in the deliver-phase done by Alice, (ii) the verification



(a) zkPoD-CR



(b) zkPoD-AS



(c) zkPoD-AS*

Fig. 12. PoD Throughput Offchain. *Suppose that N is aligned to exponents of 2.*

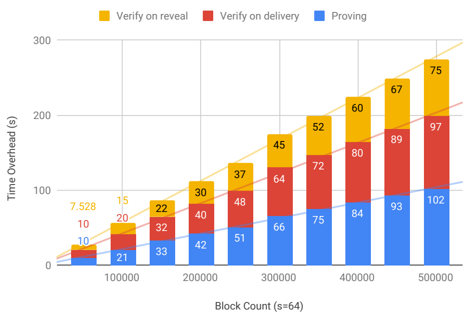
computation in the deliver-phase done by Bob, and (iii) the verification computation in the reveal-phase done by Bob. In zkPoD-CR, we can see that the work of proving is less than the work of verification of two phases (Fig. 13). In zkPoD-AS, the verification in the reveal-phase is negligible. In the deliver-phase, however, the work of proving is very close to that of verification. Therefore, the computation burden of two sides is well balanced. In zkPoD-AS*, the main computation cost is occupied by zkSNARKs generation, approximately 99% of it. In each of the three protocols, the computation grows linearly with the size of data.

Scalability and multi-threading. The majority (98%) of computation in PoD-core is parallelizable. We use option `omp_thread_num` to change the thread count and measure the time overhead for PoD-core to process the same file with different thread counts (1-12). The file size is set to 1 GiB and s is set to 64. The time overhead of three computations shrinks proportionally when the number of threads is less than 6, which is the number of CPU cores. (Fig. 14 (a1) (b1) (c1)). We can see that the data throughput of PoD-core increases almost linearly when the number of threads is less than 6 (Fig. 14 (a2) (b2) (c2)). We expect that the performance of PoD-core will has an excellent speedup if it is given more CPU cores or running on high-end GPUs.

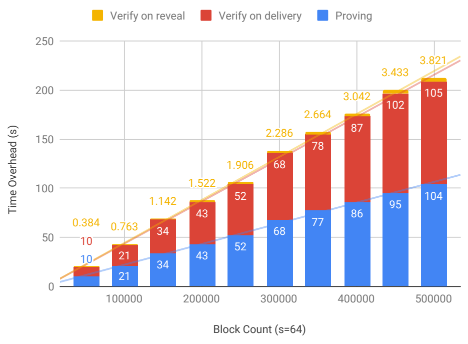
7 APPLICATIONS AND MORE RELATED WORK

7.1 Applications

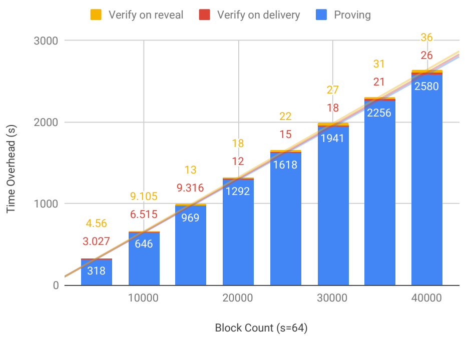
The PoD protocol can be used in any distributed applications where the system has to be aware of the events that pieces of data have been delivered from one place to another. All distributed nodes are able to verify the (PoD) proofs generated during delivery by means of smart contracts, or executable transcation scripts.



(a) zkPoD-CR



(b) zkPoD-AS

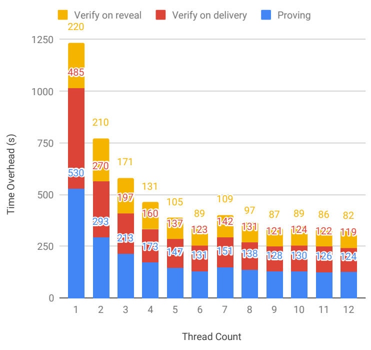


(c) zkPoD-AS*

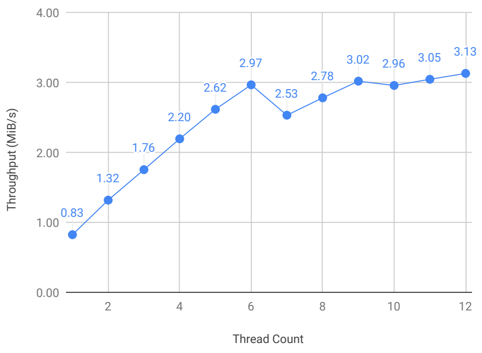
Fig. 13. Run-time Overhead of PoD Core. *Proving vs. Verify-on-delivery vs. Verify-on-reveal*

P2P network with incentives. Reportedly, P2P networks was accounted for nearly 70% of all Internet traffic 10 years before, and the propotion dropped down to 6%, of which Bittorrent took half. As the most widely used application of file sharing, the declining of BT is possibly due to the fact that it failed to provide the reliable file-sharing services for valuable but not-on-hotspot data. It is common to see that many files on p2p nodes are mutilated and miserably undownloadable. BT nodes might have been incented to keep old but valuable data so that more files would be continually stored in the network. Suppose we use PoD-based smart contracts for a user, Alice, who wants to pay and download data that hasn't been popular. Anyone who has downloaded the data, *e.g.,* Bob, would have very motivation to share any part he has. Alice may only download selected blocks of data, missing in most BT nodes, from Bob and pay in fine-grained way. By blockchain and smart contracts, anyone may publish the requirement or provide data with fair prices. PoD protocols are flexible to measure the contributions of P2P nodes by counting the blocks they relay. P2P services can easily adopt the incentive mechanism of PoD to attract more personal users to join and contribute, so as to build a highly effective and vibrant P2P file sharing network.

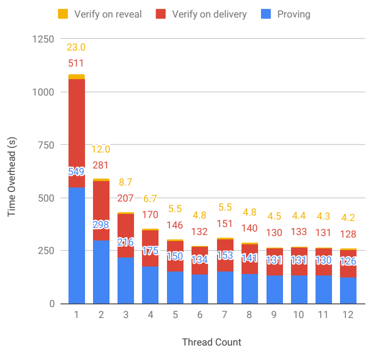
Content distribution with incentives. Content delivery network (CDN) are widely deployed to provide high availability and high performance by distributing the service spatially related to end-users. PoD protocols can be used to measure the workload of content distributing, so that every distributing transaction can be recorded on a blockchain. The trustless transactions support flexible economic incentive schemes and help to build P2P content distribution systems. IPFS is



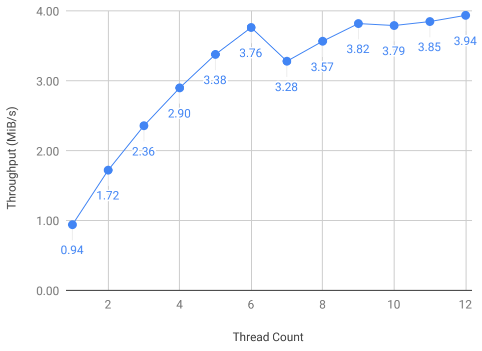
(a1) zkPoD-CR (Time Overhead)



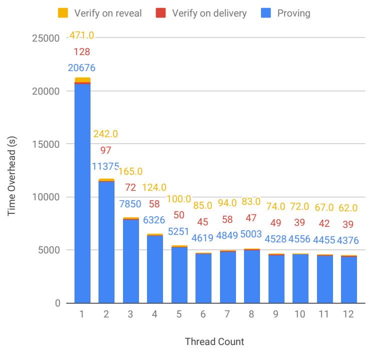
(a2) zkPoD-CR (Data Throughput)



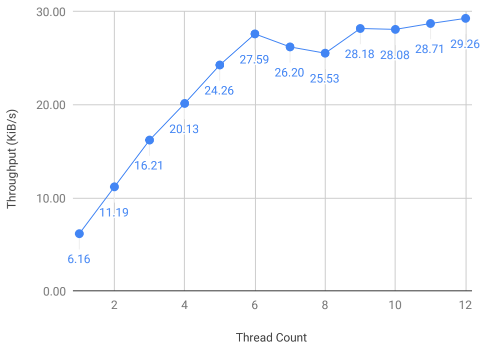
(b1) zkPoD-AS (Time Overhead)



(b2) zkPoD-AS (Data Throughput)



(c1) zkPoD-AS* (Time Overhead)



(c2) zkPoD-AS* (Data Throughput)

Fig. 14. PoD-core Performance on Multicores

a popular decentralized network filesystem. PoD protocols can be integrated into IPFS in order to build an incentive layer to encourage end-users to join the P2P network. The authenticators of data can be signed by the owner of data, then both end users and public verifiers can easily verify if the origin of the data without knowing the data content.

Contents services with automatic micropayment. Traditionally, for content providers, the cost of connecting to a centralized payment system is high. Moreover, the micropayments are infeasible unless users deposit enough money to the vendor before receiving services. PoD protocols and zkPoD system show the possibilities that a trustless payment system can be separated from content providers completely. For vendors, they can choose any payment system, or build a trustless payment system by themselves with low cost. For users, they may choose any payment systems built on blockchains which are much more secure and transparent than trusted third parties. The protocols protect users from frauds when buying digital contents. One of the big advantages of PoD protocols is supporting fine-grained data delivery, so that a buyer can make an attempt to buy only one single block or a few blocks that are randomly chosen to verify with very few coins.

Decentralized storage service. PoD protocols can be combined with proof-of-retrievabiliy (POR) so as to be integrated into decentralized storage services, like Filecoin, Storj, Siacoin *etc.* One issue of these decentralized storage systems is that users not only pay the storage spaces of miners , but also pay the retrievability. That is, more downloading, more fees. PoD protocols are natural to support retrievability with payment. Also, a zkPoD-node can also run as a standalone node providing storage serivces directly to users if it is extended with a special interface of POR queries.

Next generation of Web. In the era of classic Internet, digital contents produced by individuals are hard to monetize. When centralized content providers come, they collect digital contents without paying, re-provide contents (generated from users) back to users for free, and sell advertisements to make profits. Gradually, the centralized providers monoplized most of profits and however, the vast majority of content providers cannot get one penny from them. Even worse, the corrupted centralized providers can manipulate and impair the ecosystem. The trend is opposite to the original intention of Web. We hope that a new model of Web will be built to be more fair and friendly to individuals. zkPoD can be served as a low-level protocol to support such a new Web, where users pay for their favorite pages, and a website can get incentives from others by putting recommended links. All the browsing records are put on-chain, so that users can easily get the *real* statistics of a website, or the one which is truly valuable for them. Such an ecosystem, we believe, is friendly to newcomers with good contents, also fair to all of us.

7.2 More related work

In 2014, P.Todd and A. Taaki proposed and implemented a data trading protocol [**?**] on the Bitcoin system. The protocol is as follow: the seller uses the hash value of the data as the private key to generate the public key and the Bitcoin address. A chunk of data is encrypted by a secret that is the hash value of the public key. The buyer can pay for the chunk of data by sendingf bitcoins to the address. When the seller spends the bitcoins at the address, she will reveal the public key and thus implicitly reveal the secret. How can the buyer know if the chunk is authentic? The seller must reveal some chunks randomly at a block height in the future. The revealed chunks are chosen by the future block hash. Their approach relies on a fact that if the seller spends the bitcoin, she must provide a signature where the public key can be exported. The secret revealing and the paying are thus atomic in the sence and neither party can cheat. But the protocol isn't *strong fair* when the buyer don't pay even after the seller reveals some chunks. The seller is unwilling to reveal too much data while the buyer is still concerned with the quality of the data. Paypub protocol is atomic, by using the Bitcoin payment script to reveal keys. However, buyers cannot know if the data is what they want before paying. In PoD, data authenticators are used to ensure that any unreveald data blocks cannot be tampered. PoD also supports atomicity but with only small amounts of data blocks in the direct mode, while Paypub supports any size of data block.

S. Delgado-Segura *et al.* proposed a fair trading protocol [12] on the Bitcoin system. The idea of their protocol is that: the seller sends encrypted data to the buyer, who then builds a Bitcoin payment script revealing the encryption key in a clever way if the seller spends the payment before a specific time. To ensure buyer's interests, seller must reveal a few blocks of data (specified by the buyer) before the buyer builds the payment script. This step is indispensable because the buyer must check if the key encrypting the data is exactly the same key to be revealed by the seller. The protocol is not strong fair either since the buyer can quit the protocol after obtaining the revealed blocks. Even though the revealed part is small, a malicious user may "sybil" attack the seller by creating many buyers, each of which gets a small but different part of the data, to get the entire data eventually. Moreover, a cheating seller may mix garbage data in to gain advantages over the buyer with certain probabilities.

These two protocols are not strong fair, in the sense that a mallicious buyer or seller can gain advantages over the other, even the advantages are small. They are not suitable for being deployed on permissionless blockchain, where no identity system exists.

8 CONCLUSION AND FUTURE WORK

We have presented three PoD protocols for verifiable data delivery, PoD-AS, PoD-AS* and PoD-CR. PoD-AS and PoD-AS* achieve atomic swap like ZKCP, and PoD-CR follows the approach of Fairswap. In general, PoD protocols are low-level protocols of the infrastructure for data economy, providing trustworthy data delivery, upon which more data-driven applications can be built, *e.g.,* data processing, data storage outsourcing, data analysis and so on. Compared with many research results in the past, PoD protocols heavily rely on blockchains which act as the central parties.

We implement zkPoD system built on PoD protocols, and provides fair data exchanging, where everyone is able to participate in data exchanging without worrying about frauds. The system is wired with Ethereum, where smart contracts conduct public verification to guarantee the fairness without data leaking. It supports trustless keyword queries and the queries can be obsfucated using oblivious transfer.

To make the system more practical, there are many issues to solve. The throughput of delivered-data in PoD-AS* is still small, due to the not-so-good performance of zkSNARK proving. We hope that additional theoretic improvements can optimize zkSNARK work so as to increase the throughput of data by 2 3 orders of magnitude. The PoD protocols currently don't support generic predicates over data like in ZKCP or Fairswap. The research results recently from zero-knowledge proofs for circuit satisfiabilities, *e.g.,* zkSNARKs and bulletproofs, would be useful to support generic predicates. Another practical direction is to explore approaches to applying zero-knowledge proofs to one single data record to support more flexible queries, *e.g.,* substring matching or regular expression matching.

REFERENCES

[1] Nadarajah Asokan, Matthias Schunter, and Michael Waidner. 1996. *Optimistic protocols for fair exchange.* Citeseer.

[2] Nadarajah Asokan, Victor Shoup, and Michael Waidner. 1998. Asynchronous protocols for optimistic fair exchange. In *Proceedings. 1998 IEEE Symposium on Security and Privacy (Cat. No. 98CB36186).* IEEE, 86–99.

[3] Feng Bao, Robert H Deng, and Wenbo Mao. 1998. Efficient and practical fair exchange protocols with off-line TTP. In *Proceedings. 1998 IEEE Symposium on Security and Privacy (Cat. No. 98CB36186).* IEEE, 77–85.

[4] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Shaul Kfir, Eran Tromer, Madars Virza, and Howard Wu. [n. d.]. libsnark (2014). https://github.com/scipr-lab/libsnark.

[5] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture.. In *USENIX Security Symposium.* 781–796.

[6] Iddo Bentov and Ranjit Kumaresan. 2014. How to use bitcoin to design fair protocols. In *Annual Cryptology Conference.* Springer, 421–439.

[7] Naresh Bollam and Mr V Malsoru. 2011. Review on Data Leakage Detection. *International Journal of Engineering Research and Applications (IJERA)* 1, 3 (2011), 1088–1091.

[8] Dan Boneh, Hart William Montgomery, and Ananth Raghunathan. 2010. Algebraic pseudorandom functions with improved efficiency from the augmented cascade. In *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 131–140.

[9] Sean Bowe, Ariel Gabizon, and Ian Miers. 2017. Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model. *IACR Cryptology ePrint Archive* 2017 (2017), 1050.

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

[11] Matteo Campanelli, Rosario Gennaro, Steven Goldfeder, and Luca Nizzardo. 2017. Zero-knowledge contingent payments revisited: Attacks and payments for services. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 229–243.

[12] Sergi Delgado-Segura, Cristina Pérez-Sola, Guillermo Navarro-Arribas, and Jordi Herrera-Joancomartí. 2017. A fair protocol for data trading based on Bitcoin transactions. *Future Generation Computer Systems* (2017).

[13] Yvo Desmedt, Claude Goutier, and Samy Bengio. 1987. Special Uses and Abuses of the Fiat-Shamir Passport Protocol. In *CRYPTO*.

[14] Yevgeniy Dodis and Leonid Reyzin. 2003. Breaking and repairing optimistic fair exchange from PODC 2003. In *Proceedings of the 3rd ACM workshop on Digital rights management*. ACM, 47–54.

[15] Stefan Dziembowski, Lisa Eckey, and Sebastian Faust. 2018. Fairswap: How to fairly exchange digital goods. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 967–984.

[16] Uriel Feige, Amos Fiat, and Adi Shamir. 1988. Zero-knowledge proofs of identity. *Journal of cryptology* 1, 2 (1988), 77–94.

[17] Matthew K Franklin and Michael K Reiter. 1997. Fair exchange with a semi-trusted third party. In *ACM Conference on Computer and Communications Security*. 1–5.

[18] Benoît Garbinato and Ian Rickebusch. 2010. Impossibility results on fair exchange. *10th International Conferenceon Innovative Internet Community Systems (I2CS)–Jubilee Edition 2010–* (2010).

[19] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. 2013. Quadratic span programs and succinct NIZKs without PCPs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 626–645.

[20] Jens Groth. 2009. Linear algebra with sub-linear zero-knowledge arguments. In *Advances in Cryptology-CRYPTO 2009*. Springer, 192–208.

[21] Jens Groth. 2016. On the size of pairing-based non-interactive arguments. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 305–326.

[22] Juan Liu, Eric Bier, Aaron Wilson, John Alexis Guerra Gómez, Tomonori Honda, Kumar Sricharan, Leilani Gilpin, and Daniel Davies. 2016. Graph Analysis for Detecting Fraud, Waste, and Abuse in Healthcare Data. *AI Magazine* 37, 2 (2016), 33–46. http://www.aaai.org/ojs/index.php/aimagazine/article/view/2630

[23] Gregory Maxwell. 2016. The first successful zero-knowledge contingent payment. *Bitcoin Core, February* (2016).

[24] Silvio Micali. 2003. Simple and fast optimistic protocols for fair electronic exchange. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*. ACM, 12–19.

[25] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

[26] Henning Pagnia and Felix C Gärtner. 1999. *On the impossibility of fair exchange without a trusted third party.* Technical Report. Technical Report TUD-BS-1999-02, Darmstadt University of Technology ....

[27] Panagiotis Papadimitriou and Hector Garcia-Molina. 2009. A model for data leakage detection. In *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 1307–1310.

[28] Panagiotis Papadimitriou and Hector Garcia-Molina. 2010. Data leakage detection. *IEEE Transactions on knowledge and data engineering* 23, 1 (2010), 51–63.

[29] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2013. Pinocchio: Nearly practical verifiable computation. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 238–252.

[30] Torben Pryds Pedersen. 1991. Non-interactive and information-theoretic secure verifiable secret sharing. In *Annual International Cryptology Conference*. Springer, 129–140.

[31] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 199–212.

[32] Tuomas Sandholm and XiaoFeng Wang. 2002. (Im) possibility of safe exchange mechanism design. In *Eighteenth national conference on Artificial intelligence*. American Association for Artificial Intelligence, 338–344.

[33] Hovav Shacham and Brent Waters. 2008. Compact proofs of retrievability. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 90–107.

[34] Marie Vasek, Matthew Weeden, and Tyler Moore. 2016. Measuring the Impact of Sharing Abuse Data with Web Hosting Providers. In *Proceedings of the 2016 ACM on Workshop on Information Sharing and Collaborative Security, WISCS 2016, Vienna, Austria, October 24 - 28, 2016*. 71–80. http://dl.acm.org/citation.cfm?id=2994548

[35] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper 151 (2014).