Fair Peer-to-Peer Content Delivery via Blockchain

Songlin He[1], Yuan Lu[2(✉)], Qiang Tang[3], Guiling Wang[1], and Chase Qishi Wu[1]

[1] New Jersey Institute of Technology, Newark, NJ 07102, USA
{sh553,gwang,chase.wu}@njit.edu
[2] Institute of Software Chinese Academy of Sciences, Beijing, China
luyuan@iscas.ac.cn
[3] The University of Sydney, Sydney, Australia
qiang.tang@sydney.edu.au

Abstract. In comparison with conventional content delivery networks, peer-to-peer (p2p) content delivery is promising to save cost and handle high peak-demand, and can also complement the decentralized storage networks such as Filecoin. However, reliable p2p delivery requires proper enforcement of delivery fairness, i.e., the deliverers should be rewarded according to their in-time delivery. Unfortunately, most existing studies on delivery fairness are based on non-cooperative game-theoretic assumptions that are arguably unrealistic in the ad-hoc p2p setting.

We for the first time put forth an expressive yet still minimalist security notion for desired fair p2p content delivery, and give two efficient solutions FairDownload and FairStream via the blockchain for p2p downloading and p2p streaming scenarios, respectively. Our designs not only guarantee delivery fairness to ensure deliverers be paid (nearly) proportional to their in-time delivery but also ensure the content consumers and content providers are fairly treated. The fairness of each party can be guaranteed when the other two parties collude to arbitrarily misbehave. Moreover, the systems are efficient in the sense of attaining nearly asymptotically optimal on-chain costs and deliverer communication.

We implement the protocols and build the prototype systems atop the Ethereum Ropsten network. Extensive experiments done in LAN and WAN settings showcase their high practicality.

Keywords: Content delivery · Fairness · P2P · Blockchain application

1 Introduction

The peer-to-peer (p2p) content delivery systems are permissionless decentralized services to seamlessly replicate contents to the consumers. Typically these systems encompass a large ad-hoc network of deliverers to overcome the bandwidth bottleneck of the original content providers. In contrast to conventional pre-planned content delivery networks such as Akamai [1], p2p content delivery can crowdsource unused bandwidth resources of tremendous Internet peers, thus

E. Bertino et al. (Eds.): ESORICS 2021, LNCS 12972, pp. 348–369, 2021.
https://doi.org/10.1007/978-3-030-88418-5_17

having a wide array of benefits including robust service availability, bandwidth cost savings, and scalable peak-demand handling [2,3]. Recently, renewed attentions to p2p content delivery are gathered [2,19] due to the fast popularization of decentralized storage networks (DSNs) [5,17,34,41,42]. Indeed, most DSNs feature decentralized and robust *content storage*, but lack well-designed *content delivery* mechanisms catering for a prosperous content consumption market in the p2p setting, where the content shall not only be reliably stored but also must be always quickly *retrievable* despite potentially malicious participants [18].

The primary challenge of designing a proper delivery mechanism for complementing DSNs is to realize the strict guarantee of "fairness" against adversarial peers. In particular, it has to promise well-deserved items (e.g., retrieval of desired contents, rewards to spent bandwidth, payment for providing valid contents) to all participants [14]. Otherwise, free-riding parties can abuse the system [15,30,35] and cause rational ones to escape, eventually resulting in possible system collapse [20]. We reason as follows to distinguish two types of quintessential fairness, namely *delivery fairness* and *exchange fairness*, in the p2p content delivery setting where three parties, i.e., content *provider*, content *deliverer* and content *consumer*, are involved.

Exchange Fairness is not Delivery Fairness. Exchange fairness [4,7,10,12,28,31], specifically for digital goods (such as signatures, videos), emphasizes that one party's input shall keep *confidential* until it does learn the other party's input. Unfortunately, in the p2p content delivery setting, merely considering it becomes insufficient, because a content deliverer would expect to receive rewards proportional to the bandwidth resources that it spends. Noticeably, exchange fairness fails to capture such new desiderata related to bandwidth cost, since it allows a deliverer to receive no reward at all after transferring a huge amount of *encrypted* data to the other party, which clearly breaks the deliverer's expectation on being well-paid but does not violate exchange fairness at all.

Consider FairSwap [12] as a concrete example: the deliverer first sends the encrypted content and semantically secure digest to the consumer, then waits for a message from the consumer (via blockchain) to confirm the receival of these ciphertexts, then the deliverer can reveal his encryption key on-chain; but, in case the consumer aborts, all bandwidth used to send ciphertexts is wasted, causing no reward for deliverer. A seemingly enticing way to mitigate the above attack could be splitting the content into n smaller chunks and run FairSwap for each chunk, but the on-chain cost would grow linearly in n, resulting in prohibitive on-chain costs for large-size content like movies. Adapting other fair exchange protocols for delivery fairness would encounter similar issues. Hence, the efficient construction satisfying delivery fairness remains unclear.

Thus, to capture the special fairness required by deliverers, we formulate delivery fairness in Sect. 4, stating that deliverers can receive rewards (nearly) proportional to the contributed bandwidth for delivering data to the consumers.

Insufficiencies of Existing "Delivery Fairness". A range of literature [29,38–40] studied problems similar to delivery fairness in p2p delivery. However, to our knowledge, no one assures delivery fairness in the *cryptographic* sense

as we seek to do. In particular, they [29,38–40] were studied in the *non-cooperative game-theoretic* settings where independent attackers free ride spontaneously without communication of their strategies, and the attackers are rational with the intentions to maximize their own benefits. Therefore, it boldly ignores an adversary that intends to break the system. Unfortunately, such rational assumptions are particularly elusive to stand in ad-hoc p2p systems accessible by all malicious evils. The occurrences of massive real-world attacks in open systems [13,32] hint us how vulnerable the earlier heavy assumptions can be and further weaken the confidence of applying the prior art to real-world p2p content delivery.

Lifting for "Exchange Fairness" Between Provider and Consumer. Besides the natural delivery fairness, it is equally vital to ensure exchange fairness for providers and consumers in a basic context of p2p content delivery, especially with the end goal to complement DSNs and enable some content providers to sell contents to consumers with delegating costly delivery/storage to a p2p network. In particular, the content provider should be guaranteed to receive payments proportional to the amount of correct data learned by the consumer; vice versa, the consumer only has to pay if indeed receiving qualified content.

Naïve attempts of tuning a fair exchange protocol [4,12,28,31] into p2p content delivery can guarantee neither delivery fairness (as analyzed earlier) nor exchange fairness: simply running fair exchange protocols twice between the providers and the deliverers and between the deliverers and the consumers, respectively, would leak valuable contents, raising the threat of massive content leakage. Even worse, this idea disincentivizes the deliverers as they have to pay for the whole content before making a life by delivering the content to consumers.

Our Contributions. Overall, it remains an open problem to realize such strong fairness guarantees in p2p content delivery to protect *all* providers, deliverers, and consumers.[1] We for the first time formalize such security intuitions into a well-defined cryptographic problem on fairness and present a couple of efficient blockchain-based protocols to solve it. In sum, our contributions are:

- We formulate the problem of p2p content delivery with desired security goals, which capture the special fairness to ensure that each party (i.e., the content provider, deliverer, and consumer) is fairly treated even if the other parties arbitrarily collude or are corrupted.
- We put forth a novel delivery fairness notion dubbed verifiable fair delivery (VFD) to quantify one party's bandwidth contribution. With the instantiation of VFD, we propose the blockchain-enabled p2p content delivery protocol FairDownload, which allows: (i) the consumers can download, i.e., *view-after-delivery*, the content with minimum involvement of the provider; (ii) one-time

[1] More thorough discussions about the insufficiencies of some pertinent studies (including gradual-release based fair exchange [7,10], blockchain-based fair exchange/MPC [6,9,12,25,31], fair off-chain payment channels [11,33], and some known decentralized content delivery schemes [2,19]) are provided in the online full version [21].

contract deployment and preparation while repeatable delivery of the same content to different consumers.
To further reduce the latency in FairDownload and accommodate the streaming scenario where the consumers expect *view-while-delivery*, we propose another protocol called FairStream, such that every data chunk can be retrieved by consumers in $O(1)$ communication rounds. Though FairStream requires more involvement of an on-line content provider, the provider's online workload remains much smaller than delivering the whole content by itself.
- Both FairDownload and FairStream attain only $\tilde{O}(\eta + \lambda)$ on-chain computational costs even in the worst case, which only relates to the small chunk size parameter η and the even smaller security parameter λ. Moreover, considering the fact that $\lambda \ll \eta$, both protocols essentially realize asymptotically optimal deliverer communication complexity, as the deliverer only has to send $O(\eta + \lambda)$ bits amortized for each η-bit chunk.
- We also implement FairDownload and FairStream with making various non-trivial optimizations to reduce their critical on-chain cost.[2] Extensive experiments in WAN and LAN settings showcase their real-world applicability.

2 Preliminaries

Notations. Let $[n]$ denote $\{1, \ldots, n\}$, $[a, b]$ denote $\{a, \ldots, b\}$, $x||y$ denote concatenating x and y, $\leftarrow_{\$}$ denote uniformly random sampling, and $\preceq$ denote the prefix relationship.

Global ledger. It provides the primitive of cryptocurrency that can deal with "coin" transfers transparently. Specifically, each entry of the dictionary $\mathsf{ledger}[\mathcal{P}_i]$ records the balance of the party $\mathcal{P}_i$, and is global (which means it is accessible by all system participants including the adversary). Moreover, the global dictionary ledger can be a subroutine of the so-called *smart contract* to transact "coins" to a designated party when some conditions are met.

Cryptographic Primitives. We consider: (i) a hash function $\mathcal{H} : \{0,1\}^* \rightarrow \{0,1\}^\lambda$ modeled as a random oracle; (ii) a *semantically secure* (fixed-length) symmetric encryption scheme consisting of $(\mathsf{SE.KGen}, \mathsf{SEnc}, \mathsf{SDec})$; (iii) an *existential unforgeability under chosen message attack (EU-CMA)* secure digital signature scheme consisting of $(\mathsf{SIG.KGen}, \mathsf{Sign}, \mathsf{Verify})$; (iv) a Merkle tree scheme of $(\mathsf{BuildMT}, \mathsf{GenMTP}, \mathsf{VerifyMTP})$; (v) a specific verifiable public key encryption scheme VPKE consisting of $(\mathsf{VPKE.KGen}, \mathsf{VEnc}, \mathsf{VDec}, \mathsf{ProvePKE}, \mathsf{VerifyPKE})$ allowing the decryptor to produce a proof besides the plaintext, thus attesting correct decryption in a zero-knowledge fashion, i.e., without leaking private key [8].

3 Warm-Up: Verifiable Fair Delivery

We first warm up and set forth a building block termed *verifiable fair delivery* (VFD), which enables an honest verifier to validate that a sender indeed transfers

[2] Code availability: https://github.com/Blockchain-World/FairThunder.git.

some amount of data to a receiver. It later acts as a key module in the fair p2p content delivery protocol (in Sect. 5). The high level idea of VFD lies in: a receiver needs to send back a signed "receipt" in order to acknowledge a sender's bandwidth contribution and continuously receives the next data chunk. Consider the data chunks of same size η are transferred *sequentially* starting from the first chunk, the sender can always use the latest receipt containing the chunk index to prove to a verifier about the contribution. Intuitively the sender *at most* wastes bandwidth of transferring one chunk.

Syntax. The VFD protocol is among an interactive poly-time Turing-machine (ITM) sender $\mathcal{S}$, an ITM receiver $\mathcal{R}$, and a non-interactive Turing-machine verifier $\mathcal{V}$, and follows the syntax:

- **Sender.** $\mathcal{S}$ can be activated by an interface $\mathcal{S}.\mathsf{send}()$ with inputting a sequence of n data chunks and their corresponding validation strings, denoted by $((c_1, \sigma_{c_1}), \ldots, (c_n, \sigma_{c_n}))$, and there exists an efficient and global predicate $\Psi(i, c_i, \sigma_{c_i}) \rightarrow \{0, 1\}$ to check whether c_i is the i-th valid chunk due to σ_{c_i}; once activated, $\mathcal{S}$ interacts with $\mathcal{R}$ and opens an interface $\mathcal{S}.\mathsf{prove}()$ that can be invoked to generate a proof π indicating the number of sent chunks;
- **Receiver.** $\mathcal{R}$ can be activated by an interface $\mathcal{R}.\mathsf{recv}()$ with the input of the global predicate $\Psi(\cdot)$ to interact with $\mathcal{S}$, and outputs a sequence of $((c_1, \sigma_{c_1}), \ldots, (c_{n'}, \sigma_{c_{n'}}))$, where $n' \in [n]$ and every (c_i, σ_{c_i}) is valid due to $\Psi(\cdot)$;
- **Verifier.** $\mathcal{V}$ takes as input the proof π generated by $\mathcal{S}.\mathsf{prove}()$, and outputs an integer $\mathsf{ctr} \in \{0, \cdots, n\}$.

Security. The VFD protocol must satisfy the following security requirements:

- **Termination.** If at least one of $\mathcal{S}$ and $\mathcal{R}$ is honest, the VFD protocol terminates within at most $2n$ rounds, where n is the number of content chunks.
- **Completeness.** If $\mathcal{S}$ and $\mathcal{R}$ are both honest and activated, after $2n$ rounds, $\mathcal{S}$ is able to generate a proof π that can be verified by $\mathcal{V}$ to output $\mathsf{ctr} = n$, while $\mathcal{R}$ can output $((c_1, \sigma_{c_1}), \ldots, (c_n, \sigma_{c_n}))$, which is same to $\mathcal{S}$'s input.
- **Verifiable η delivery fairness**. When one of $\mathcal{S}$ and $\mathcal{R}$ maliciously aborts, VFD shall satisfy the following delivery fairness requirements:
  - *Verifiable delivery fairness against $\mathcal{S}^*$*. The honest receiver $\mathcal{R}$ will always receive the valid sequence $(c_1, \sigma_{c_1}), \ldots, (c_{\mathsf{ctr}}, \sigma_{c_{\mathsf{ctr}}})$ if the corrupted $\mathcal{S}^*$ can produce the proof π that enables $\mathcal{V}$ to output ctr.
  - *Verifiable delivery fairness against $\mathcal{R}^*$*. The honest sender $\mathcal{S}$ can always generate a proof π, which enables $\mathcal{V}$ to output *at least* $(\mathsf{ctr} - 1)$ if the corrupted $\mathcal{R}^*$ receives the valid sequence $(c_1, \sigma_{c_1}), \ldots, (c_{\mathsf{ctr}}, \sigma_{c_{\mathsf{ctr}}})$. At most $\mathcal{S}$ wastes bandwidth for delivering one content chunk of η-bit size.

VFD protocol Π_{VFD}. We consider the authenticated setting where the sender $\mathcal{S}$ and the receiver $\mathcal{R}$ have generated public-private key pairs $(pk_{\mathcal{S}}, sk_{\mathcal{S}})$ and $(pk_{\mathcal{R}}, sk_{\mathcal{R}})$, respectively, and they have announced the public keys to bind to themselves. Then, VFD with the global predicate $\Psi(\cdot)$ can be realized by Π_{VFD}

hereunder among $\mathcal{S}$, $\mathcal{R}$ and $\mathcal{V}$ against probabilistic poly-time (P.P.T.) and static adversary in the *stand-alone* setting[3] with the synchronous network assumption:

- **Construction of $\mathcal{S}$.** The sender, after activated via $\mathcal{S}.\mathsf{send}()$ with the input $((c_1, \sigma_{c_1}), \ldots, (c_n, \sigma_{c_n}))$, $pk_{\mathcal{S}}$ and $pk_{\mathcal{R}}$, starts a timer $\mathcal{T}_{\mathcal{S}}$ lasting two synchronous rounds, initializes a variable $\pi_{\mathcal{S}} := null$, and executes as follows:
  - For each $i \in [n]$: sends $(\mathsf{deliver}, i, c_i, \sigma_{c_i})$ to $\mathcal{R}$, and waits for $(\mathsf{receipt}, i, \sigma^i_{\mathcal{R}})$ from $\mathcal{R}$. If $\mathcal{T}_{\mathcal{S}}$ expires before receiving the $\mathsf{receipt}$, breaks the iteration; otherwise $\mathcal{S}$ verifies whether $\mathsf{Verify}(\mathsf{receipt}||i||pk_{\mathcal{R}}||pk_{\mathcal{S}}, \sigma^i_{\mathcal{R}}, pk_{\mathcal{R}}) \equiv 1$ or not, if *true*, resets $\mathcal{T}_{\mathcal{S}}$, outputs $\pi_{\mathcal{S}} := (i, \sigma^i_{\mathcal{R}})$, and continues to run the next iteration (i.e., increasing i by one); if *false*, breaks the iteration;
  - Upon $\mathcal{S}.\mathsf{prove}()$ is invoked, it returns $\pi_{\mathcal{S}}$ as the VFD proof and halts.
- **Construction of $\mathcal{R}$.** The receiver, after activated via $\mathcal{R}.\mathsf{recv}()$ with the input $pk_{\mathcal{S}}$ and $(pk_{\mathcal{R}}, sk_{\mathcal{R}})$, starts a timer $\mathcal{T}_{\mathcal{R}}$ lasting two synchronous rounds and operates as: for each $j \in [n]$: $\mathcal{R}$ waits for $(\mathsf{deliver}, j, c_j, \sigma_{c_j})$ from $\mathcal{S}$ and halts if $\mathcal{T}_{\mathcal{R}}$ expires before receiving the $\mathsf{deliver}$ message; otherwise $\mathcal{R}$ verifies whether $\Psi(j, c_j, \sigma_{c_j}) \equiv 1$ or not; if *true*, resets $\mathcal{T}_{\mathcal{R}}$, outputs (c_j, σ_{c_j}), and sends $(\mathsf{receipt}, i, \sigma^i_{\mathcal{R}})$ to $\mathcal{S}$ where $\sigma^i_{\mathcal{R}} \leftarrow \mathsf{Sign}(\mathsf{receipt}||i||pk_{\mathcal{R}}||pk_{\mathcal{S}}, sk_{\mathcal{R}})$, halts if *false*. Note that $\Psi(\cdot)$ is efficient as it just performs a signature verification.
- **Construction of $\mathcal{V}$.** Upon the input $\pi_{\mathcal{S}}$, the verifier $\mathcal{V}$ parses it into $(\mathsf{ctr}, \sigma^{\mathsf{ctr}}_{\mathcal{R}})$, and checks whether $\mathsf{Verify}(\mathsf{receipt}||\mathsf{ctr}||pk_{\mathcal{R}}||pk_{\mathcal{S}}, \sigma^{\mathsf{ctr}}_{\mathcal{R}}, pk_{\mathcal{R}}) \equiv 1$ or not; if *true*, it outputs ctr, or else outputs 0. Recall that Verify is to verify signatures.

The following lemma states the security of the above VFD protocol, the detailed proof of which is presented in the online full version [21].

Lemma 1. *In the synchronous authenticated network and stand-alone setting, Π_{VFD} satisfies termination, completeness and the verifiable η delivery fairness against static P.P.T. adversary corrupting one of the sender and the receiver.*

4 Formalizing P2P Content Delivery

4.1 System Model

Participating Parties. We consider the following entities (i.e., interactive Turing machines by cryptographic convention) in the context of p2p content delivery:

- *Content Provider $\mathcal{P}$* is an entity that owns the original content m composed of n chunks,[4] satisfying a public known predicate $\phi(\cdot)$,[5] and $\mathcal{P}$ is willing to

[3] To defend against *replay* attack in concurrent sessions, it is trivial to let the authenticated messages include a session id *sid* field, which, for example, can be instantiated by the hash of the transferred data identifier root_m, the involved parties' addresses and an increasing-only nonce, namely $sid := \mathcal{H}(\mathsf{root}_m||\mathcal{V}\_address||pk_{\mathcal{S}}||pk_{\mathcal{R}}||nonce)$.

[4] Remark that the content m is *dividable* in the sense that each chunk is *independent* to other chunks, e.g., the chunk is a small 10-s video fragment.

[5] Throughout this paper, we consider ϕ is in the form of $\phi(m) = [\mathsf{root}(\mathsf{BuildMT}(m)) \equiv \mathsf{root}_m]$, where root is the Merkle tree root of m. In practice, it can be acquired from a semi-trusted third party, e.g., VirusTotal [23] or BitTorrent forum sites [28].

sell to the users of interest. Also, $\mathcal{P}$ would like to delegate the delivery of m to a deliverer with promise to pay $\ddot{\mathbb{B}}_{\mathcal{P}}$ for each successfully delivered chunk.
- *Content Deliverer $\mathcal{D}$* contributes its *idle* bandwidth resources to deliver the content on behalf of the provider $\mathcal{P}$ and would receive the payment proportional to the amount of delivered data.
- *Content Consumer $\mathcal{C}$* is an entity that would pay $\ddot{\mathbb{B}}_{\mathcal{C}}$ for each chunk in the content m by interacting with $\mathcal{P}$ and $\mathcal{D}$.

Adversary $\mathcal{A}$. We consider the adversary with the following standard abilities [24]: (i) *Static corruptions:* $\mathcal{A}$ can corrupt some parties only before the protocol executions; (ii) *Computationally bounded:* $\mathcal{A}$ is restricted to P.P.T. algorithms; (iii) *Synchronous authenticated channel:* it describes the ability of $\mathcal{A}$ on controlling communications. W.l.o.g., we consider a global clock in the system, and $\mathcal{A}$ can delay the messages up to a clock round [25,27].

Arbiter Smart Contract $\mathcal{G}$. The system is in a hybrid model with oracle access to an arbiter smart contract $\mathcal{G}$, which is a stateful ideal functionality that leaks all its internal states to the adversary $\mathcal{A}$ and all parties, and can check some immutable conditions to transact "coins" over the cryptocurrency ledger, thus "mimicking" the contracts in real life transparently. In practice, the contract can be instantiated via real-world blockchains such as Ethereum [43]. Description of $\mathcal{G}$ in the paper follows the conventional pseudo-code notations in [27].

4.2 Design Goals

Syntax. A fair p2p content delivery protocol $\Pi = (\mathcal{P}, \mathcal{D}, \mathcal{C})$ is a tuple of three P.P.T. interactive Turing machines (ITMs) consisting of two explicit phases:

- *Preparation phase.* The provider $\mathcal{P}$ takes as input public parameters and the content $m = (m_1, \ldots, m_n) \in \{0,1\}^{\eta \times n}$ that satisfies $\phi(m) \equiv 1$, where η is chunk size in bit and n is the number of chunks, and it outputs some auxiliary data, e.g., encryption keys; the deliverer $\mathcal{D}$ takes as input public parameters and outputs some auxiliary data, e.g., encrypted content; the consumer $\mathcal{C}$ does not involve in this phase. Note $\mathcal{P}$ deposits a budget of $n \cdot \ddot{\mathbb{B}}_{\mathcal{P}}$ in ledger to incentivize $\mathcal{D}$ so it can *minimize* bandwidth usage in the next phase.
- *Delivery phase.* The provider $\mathcal{P}$ and the deliverer $\mathcal{D}$ take as input their auxiliary data obtained in the preparation phase, respectively, and they would receive the deserved payment; the consumer $\mathcal{C}$ takes as input public parameters and outputs the content m with $\phi(m) \equiv 1$. Note $\mathcal{C}$ has a budget of $n \cdot \ddot{\mathbb{B}}_{\mathcal{C}}$ in ledger to "buy" the content m satisfying $\phi(m) \equiv 1$, where $\ddot{\mathbb{B}}_{\mathcal{C}} > \ddot{\mathbb{B}}_{\mathcal{P}}$.

Properties. Besides, the protocol shall guarantee the following properties.

Completeness. For any content predicate $\phi(m) = [\mathsf{root}(\mathsf{BuildMT}(m)) \equiv \mathsf{root}_m]$, conditioned on $\mathcal{P}, \mathcal{D}$ and $\mathcal{C}$ are all honest, the protocol Π attains: (i) $\mathcal{C}$ obtains the content m satisfying $\phi(m) \equiv 1$, and the balance of $\mathsf{ledger}[\mathcal{C}]$ decreases by $n \cdot \ddot{\mathbb{B}}_{\mathcal{C}}$; (ii) $\mathcal{D}$ receives the payment $n \cdot \ddot{\mathbb{B}}_{\mathcal{P}}$ over the global ledger; (iii) $\mathcal{P}$ gets the payment $n \cdot (\ddot{\mathbb{B}}_{\mathcal{C}} - \ddot{\mathbb{B}}_{\mathcal{P}})$, as it receives $n \cdot \ddot{\mathbb{B}}_{\mathcal{C}}$ from $\mathcal{C}$ and pays $n \cdot \ddot{\mathbb{B}}_{\mathcal{P}}$ to $\mathcal{D}$.

Fairness. The protocol Π should satisfy the following fairness requirements:

- *Consumer Fairness.* The honest consumer $\mathcal{C}$ is ensured that: the $\mathsf{ledger}[\mathcal{C}]$ decreases by $\ell \cdot \ddot{\mathbb{B}}_{\mathcal{C}}$ only if $\mathcal{C}$ receives a sequence of chunks $(m_1, \ldots, m_\ell) \preceq m$ where $\phi(m) \equiv 1$, i.e., $\mathcal{C}$ pays proportional to valid chunks it *de facto* receives.
- *Deliverer η-Fairness.* The honest deliverer $\mathcal{D}$ is assured that: if $\mathcal{D}$ sent overall $O(\ell \cdot \eta + 1)$ bits during the protocol, $\mathcal{D}$ should *at least* obtain the payment of $(\ell - 1) \cdot \ddot{\mathbb{B}}_{\mathcal{P}}$. In other words, the unpaid delivery is bounded by $O(\eta)$ bits.
- *Provider η-Fairness.* The honest provider $\mathcal{P}$ is guaranteed that: if $\mathcal{A}$ can output $\eta \cdot \ell$ bits prefixed in the content m, $\mathcal{P}$ should *at least* receive $(\ell-1) \cdot (\ddot{\mathbb{B}}_{\mathcal{C}} - \ddot{\mathbb{B}}_{\mathcal{P}})$ net income. Intuitively, $\mathcal{P}$ is ensured that *at most* $O(\eta)$-bit content are revealed without being well paid.

Confidentiality Against Deliverer. A malicious $\mathcal{D}^*$ corrupted by $\mathcal{A}$ cannot output $\mathcal{P}$'s original content in a delivery session even after receiving all protocol scripts and all internal states leaked by the contract. Note that confidentiality is not captured by fairness, as it is trivial to see that a protocol satisfying fairness might not have confidentiality: upon all payments are cleared and the consumer receives the whole content, the protocol lets the consumer send the content to the deliverer.

Timeliness. When at least one of the parties $\mathcal{P}$, $\mathcal{D}$ and $\mathcal{C}$ is honest (i.e., others are corrupted by $\mathcal{A}$), the honest ones are guaranteed to halt in $O(n)$ synchronous rounds where n is the number of content chunks. At completion or abortion of the protocol, the fairness and confidentiality properties are always guaranteed.

Non-trivial Efficiency. We require the necessary efficiency to rule out possible trivial solutions: (i) the messages sent to $\mathcal{G}$ from honest parties are uniformly bounded by $\tilde{O}(1)$ bits, which excludes a trivial way of directly delivering content via smart contract; (ii) in the delivery phase, the messages sent by honest $\mathcal{P}$ are uniformly bounded by $n \cdot \lambda$ bits, where λ is a small cryptographic parameter such that $n \cdot \lambda$ is much smaller than the content size $|m|$. This makes $\mathcal{P}$ to save bandwidth after preparation phase and excludes the idea of delivering by itself.

Remarks About Definition. It is worth noticing that: (i) the predicate $\phi(\cdot)$ is a public parameter; (ii) our fairness requirements have implied the case of corrupting one party of $\mathcal{P}$, $\mathcal{D}$ and $\mathcal{C}$ instead of two since $\mathcal{A}$ can always let some corrupted one follow the original protocol; (iii) we do not consider the case that all parties are corrupted; (iv) the deliverer and the provider might lose well-deserved payment, but *at most* for one chunk, i.e., the level of unfairness is strictly bounded; (v) though we focus on the case of a *single* content deliverer, our formalism and design can be extended to capture *multiple* deliverers, e.g., by cutting the content into multiple pieces and each piece is delegated to a distinct deliverer, which forms a future extension; (vi) after the one-time preparation phase, the delivery phase becomes repeatable.

In addition, one might wonder that a probably corrupted content provider fails in the middle of a transmission, causing that the consumer does not get the entire content but has to pay a lot. Nevertheless, this is not a serious worry in

the peer-to-peer content delivery setting that aims to complement decentralized content storage networks, because there essentially are a large number of deliverers, and at least some of them can be honest. As such, if a consumer encounters failures in the middle of delivery, it can iteratively ask another deliverer to start a new session to fetch the remaining undelivered chunks. Moreover, our constructions in Sect. 5 and Sect. 6 allow the consumers to fetch the content from any specific chunk instead of always starting from the first chunk, the expense of which would be nearly proportional to the actual number of retrieved chunks.

5 FairDownload: Fair P2P Downloading

5.1 FairDownload Overview

At a high level, FairDownload is constructed around the module of verifiable fair delivery (VFD) and operates in *Prepare*, *Deliver* and *Reveal* phases, as illustrated in Fig. 1. The core ideas are: initially the provider $\mathcal{P}$ encrypts each chunk, signs the encrypted chunks and delegates to the deliverer $\mathcal{D}$; the consumer $\mathcal{C}$ and $\mathcal{D}$ can run an instance of VFD, where the predicate $\Psi(\cdot)$ ensures each delivered chunk is correctly signed by $\mathcal{P}$; the arbiter smart contract $\mathcal{G}_d^{\mathsf{ledger}}$ (abbr. $\mathcal{G}_d$) shown in Fig. 2 can invoke VFD verifier to check the VFD proof and realize whether $\mathcal{D}$ indeed delivers ctr chunks to $\mathcal{C}$; $\mathcal{P}$ then presents to reveal the minimum (i.e., a short $\tilde{O}(\lambda)$-bit message) number of elements (via our proposed *structured key derivation scheme* composed of Algorithms 1, 2 and 4) on-chain, so that $\mathcal{C}$ can use a small string to recover the decryption keys for all ctr chunks; the revealed on-chain elements are encrypted by the consumer's public key to ensure confidentiality against malicious $\mathcal{D}$; In case of dispute, $\mathcal{C}$ can complain to the contract $\mathcal{G}_d$ via a short $O(\eta + \lambda)$-bit message to "prove" the error of decrypted chunk and get refund, the on-chain cost of verifying which is $O(\log n)$ at worst.

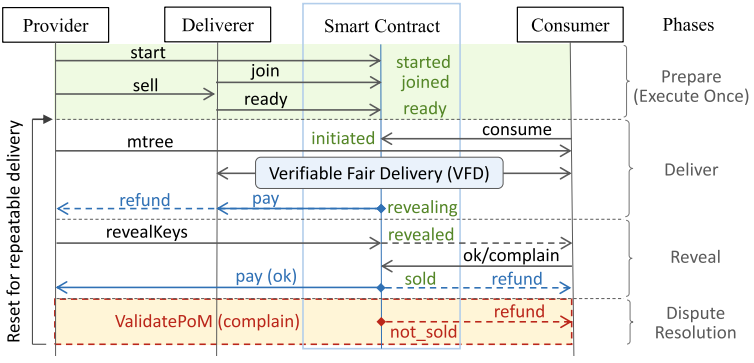


Fig. 1. The execution flow of FairDownload protocol Π_{FD}.

The Arbiter Contract Functionality $\mathcal{G}_d^{\mathsf{ledger}}$ for P2P Downloading

The arbiter contract $\mathcal{G}_d$ has access to the ledger, and it interacts with the provider $\mathcal{P}$, the deliverer $\mathcal{D}$, the consumer $\mathcal{C}$ and the adversary $\mathcal{A}$. It locally stores the times of repeatable delivery θ, the number of content chunks n, the content digest root_m, the price $\mathbb{B}_{\mathcal{P}}$, $\mathbb{B}_{\mathcal{C}}$ and $\mathbb{B}_{\mathsf{pf}}$, the number of delivered chunks ctr (initialized as 0), addresses $pk_{\mathcal{P}}, pk_{\mathcal{D}}, pk_{\mathcal{C}}, vpk_{\mathcal{C}}$, revealed keys' hash erk_{hash}, state Σ and three timers $\mathcal{T}_{\mathsf{round}}$ (implicitly), $\mathcal{T}_{\mathsf{deliver}}$, and $\mathcal{T}_{\mathsf{dispute}}$.

Phase 1: Prepare

- On receive $(\mathsf{start}, pk_{\mathcal{P}}, \mathsf{root}_m, \theta, n, \mathbb{B}_{\mathcal{P}}, \mathbb{B}_{\mathcal{C}}, \mathbb{B}_{\mathsf{pf}})$ from $\mathcal{P}$:
  - assert $\mathsf{ledger}[\mathcal{P}] \geq (\theta \cdot (n \cdot \mathbb{B}_{\mathcal{P}} + \mathbb{B}_{\mathsf{pf}})) \wedge \Sigma \equiv \emptyset$
  - store $pk_{\mathcal{P}}, \mathsf{root}_m, \theta, n, \mathbb{B}_{\mathcal{P}}, \mathbb{B}_{\mathcal{C}}, \mathbb{B}_{\mathsf{pf}}$
  - let $\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] - \theta \cdot (n \cdot \mathbb{B}_{\mathcal{P}} + \mathbb{B}_{\mathsf{pf}})$ and $\Sigma := \mathsf{started}$
  - send $(\mathsf{started}, pk_{\mathcal{P}}, \mathsf{root}_m, \theta, n, \mathbb{B}_{\mathcal{P}}, \mathbb{B}_{\mathcal{C}}, \mathbb{B}_{\mathsf{pf}})$ to all entities
- On receive $(\mathsf{join}, pk_{\mathcal{D}})$ from $\mathcal{D}$:
  - assert $\Sigma \equiv \mathsf{started}$
  - store $pk_{\mathcal{D}}$ and let $\Sigma := \mathsf{joined}$
  - send $(\mathsf{joined}, pk_{\mathcal{D}})$ to all entities
- On receive $(\mathsf{prepared})$ from $\mathcal{D}$:
  - assert $\Sigma \equiv \mathsf{joined}$, and let $\Sigma := \mathsf{ready}$
  - send (ready) to all entities

Phase 2: Deliver

- On receive $(\mathsf{consume}, pk_{\mathcal{C}}, vpk_{\mathcal{C}})$ from $\mathcal{C}$:
  - assert $\theta > 0$
  - assert $\mathsf{ledger}[\mathcal{C}] \geq n \cdot \mathbb{B}_{\mathcal{C}} \wedge \Sigma \equiv \mathsf{ready}$
  - store $pk_{\mathcal{C}}$, $vpk_{\mathcal{C}}$ and let $\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] - n \cdot \mathbb{B}_{\mathcal{C}}$
  - start a timer $\mathcal{T}_{\mathsf{deliver}}$ and let $\Sigma := \mathsf{initiated}$
  - send $(\mathsf{initiated}, pk_{\mathcal{C}}, vpk_{\mathcal{C}})$ to all entities
- On receive $(\mathsf{delivered})$ from $\mathcal{C}$ or $\mathcal{T}_{\mathsf{deliver}}$ times out:
  - assert $\Sigma \equiv \mathsf{initiated}$
  - send $(\mathsf{getVFDProof})$ to $\mathcal{D}$, and wait for two rounds to receive $(\mathsf{receipt}, i, \sigma_{\mathcal{C}}^i)$, then execute $\mathsf{verifyVFDProof}()$ to let $\mathsf{ctr} := i$, and then assert $0 \leq \mathsf{ctr} \leq n$
  - let $\mathsf{ledger}[\mathcal{D}] := \mathsf{ledger}[\mathcal{D}] + \mathsf{ctr} \cdot \mathbb{B}_{\mathcal{P}}$
  - let $\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] + (n - \mathsf{ctr}) \cdot \mathbb{B}_{\mathcal{P}}$
  - store ctr, let $\Sigma := \mathsf{revealing}$, and send $(\mathsf{revealing}, \mathsf{ctr})$ to all entities

Phase 3: Reveal

- On receive $(\mathsf{revealKeys}, erk)$ from $\mathcal{P}$:
  - assert $\Sigma \equiv \mathsf{revealing}$
  - store erk (essentially erk's hash) and start a timer $\mathcal{T}_{\mathsf{dispute}}$
  - let $\Sigma := \mathsf{revealed}$
  - send $(\mathsf{revealed}, erk)$ to all entities
- Upon $\mathcal{T}_{\mathsf{dispute}}$ times out:
  - assert $\Sigma \equiv \mathsf{revealed}$ and current time $\mathcal{T} \geq \mathcal{T}_{\mathsf{dispute}}$
  - $\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] + \mathsf{ctr} \cdot \mathbb{B}_{\mathcal{C}} + \mathbb{B}_{\mathsf{pf}}$
  - $\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] + (n - \mathsf{ctr}) \cdot \mathbb{B}_{\mathcal{C}}$
  - let $\Sigma := \mathsf{sold}$ and send (sold) to all entities

▷ Below is the dispute resolution

- On receive $(\mathsf{wrongRK})$ from $\mathcal{C}$ before $\mathcal{T}_{\mathsf{dispute}}$ times out:
  - assert $\Sigma \equiv \mathsf{revealed}$ and current time $\mathcal{T} < \mathcal{T}_{\mathsf{dispute}}$
  - if $(\mathsf{ValidateRKeys}(n, \mathsf{ctr}, erk) \equiv false)$:
    - let $\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] + n \cdot \mathbb{B}_{\mathcal{C}} + \mathbb{B}_{\mathsf{pf}}$
    - let $\Sigma := \mathsf{not\_sold}$ and send $(\mathsf{not\_sold})$ to all entities
- On receive $(\mathsf{PoM}, i, j, c_i, \sigma_{c_i}, \mathcal{H}(m_i), \pi_{\mathsf{MT}}^i, rk, erk, \pi_{\mathsf{VD}})$ from $\mathcal{C}$ before $\mathcal{T}_{\mathsf{dispute}}$ times out:
  - assert $\Sigma \equiv \mathsf{revealed}$ and current time $\mathcal{T} < \mathcal{T}_{\mathsf{dispute}}$
  - invoke the $\mathsf{ValidatePoM}(i, j, c_i, \sigma_{c_i}, \mathcal{H}(m_i), \pi_{\mathsf{MT}}^i, rk, erk, \pi_{\mathsf{VD}})$ subroutine, if $true$ is returned:
    - let $\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] + n \cdot \mathbb{B}_{\mathcal{C}} + \mathbb{B}_{\mathsf{pf}}$
    - let $\Sigma := \mathsf{not\_sold}$ and send $(\mathsf{not\_sold})$ to all entities

▷ Reset to the ready state for repeatable delivery

- On receive (reset) from $\mathcal{P}$:
  - assert $\Sigma \equiv \mathsf{sold}$ or $\Sigma \equiv \mathsf{not\_sold}$
  - set ctr, $\mathcal{T}_{\mathsf{deliver}}$, $\mathcal{T}_{\mathsf{dispute}}$ as 0
  - nullify $pk_{\mathcal{C}}$ and $vpk_{\mathcal{C}}$
  - let $\theta := \theta - 1$, and $\Sigma := \mathsf{ready}$
  - send (ready) to all entities

Fig. 2. The arbiter contract functionality $\mathcal{G}_d^{\mathsf{ledger}}$ for downloading. "Sending to all entities" captures that the smart contract is transparent to the public.

5.2 Π_{FD}: FairDownload Protocol

Now we present the fair p2p downloading protocol Π_{FD}. The session id *sid* and the content digest root_m are omitted since they remain same in a delivery session.

Phase I for Prepare. The parties $\mathcal{P}$ and $\mathcal{D}$ interact with $\mathcal{G}_d$ in this phase as:

- The provider $\mathcal{P}$ with $(pk_{\mathcal{P}}, sk_{\mathcal{P}})$ deploys contracts and starts[6] Π_{FD} by sending $(\mathsf{start}, pk_{\mathcal{P}}, \mathsf{root}_m, \theta, n, \ddot{\mathbb{B}}_{\mathcal{P}}, \ddot{\mathbb{B}}_{\mathcal{C}}, \ddot{\mathbb{B}}_{\mathsf{pf}})$ to $\mathcal{G}_d$, where root_m is the content digest in the form of Merkle tree root, θ indicates how many times of repeatable delivery are allowed for this contract, n is the number of content chunks[7], $\ddot{\mathbb{B}}_{\mathcal{P}}, \ddot{\mathbb{B}}_{\mathcal{C}}, \ddot{\mathbb{B}}_{\mathsf{pf}} \in \mathbb{N}$ are price parameters, and $\ddot{\mathbb{B}}_{\mathsf{pf}}$ is the *penalty fee*[8] in a delivery session to discrouage the misbehavior from $\mathcal{P}$.
- Upon $\Sigma \equiv \mathsf{joined}$, the provider $\mathcal{P}$ would: (i) randomly sample a master key mk, run Algorithm 1 to get the key derivation tree KT, and store mk and KT locally; (ii) use the n leaf nodes of KT to encrypt $(m_1, \ldots, m_n)$ to get $(c_1, \ldots, c_n)$; (iii) then sign the encrypted chunks to obtain a sequence of ciphertext-signature pairs $((c_1, \sigma_{c_1}), \ldots, (c_n, \sigma_{c_n}))$; compute content's Merkle tree MT and sign MT to obtain the signature $\sigma_{\mathcal{P}}^{\mathsf{MT}}$; locally store $(\mathsf{MT}, \sigma_{\mathcal{P}}^{\mathsf{MT}})$ and then send $(\mathsf{sell}, ((c_1, \sigma_{c_1}), \cdots, (c_n, \sigma_{c_n})))$ to $\mathcal{D}$ off-chain; (iv) wait for (ready) from $\mathcal{G}_d$ to enter the next phase.
- The deliverer $\mathcal{D}$ with $(pk_{\mathcal{D}}, sk_{\mathcal{D}})$ would: (i) upon receiving the $(\mathsf{started})$ message from $\mathcal{G}_d$, send $(\mathsf{join}, pk_{\mathcal{D}})$ to $\mathcal{G}_d$; (ii) wait for the (sell) message from $\mathcal{P}$, then verify each pair (c_i, σ_{c_i}). If valid, send $(\mathsf{prepared})$ to $\mathcal{G}_d$, and store $((c_1, \sigma_{c_1}), \cdots, (c_n, \sigma_{c_n}))$ locally, otherwise halt; (iii) wait for (ready) from $\mathcal{G}_d$ to enter the next phase.

Upon the completion of the above execution, $\mathcal{P}$ owns a master key mk, the key derivation tree KT, and the Merkle tree MT, while $\mathcal{D}$ receives the delegated encrypted content chunks and is ready to deliver the content to consumers.

Phase II for Deliver. The parties $\mathcal{C}$, $\mathcal{P}$, and $\mathcal{D}$ interact with $\mathcal{G}_d$ as follows:

- The consumer $\mathcal{C}$ with $(pk_{\mathcal{C}}, sk_{\mathcal{C}})$ and $(vpk_{\mathcal{C}}, vsk_{\mathcal{C}}) \leftarrow \mathsf{VPKE.KGen}(1^{\lambda})$ would: (i) assert $\Sigma \equiv \mathsf{ready}$, and send $(\mathsf{consume}, pk_{\mathcal{C}}, vpk_{\mathcal{C}})$ to $\mathcal{G}_d$; (ii) upon receiving $(\mathsf{mtree}, \mathsf{MT}, \sigma_{\mathcal{P}}^{\mathsf{MT}})$ from $\mathcal{P}$, verify the signature $\sigma_{\mathcal{P}}^{\mathsf{MT}}$ and check whether $\mathsf{root}(\mathsf{MT}) \equiv \mathsf{root}_m$ or not, if both hold, store MT and then activate the receiver $\mathcal{R}$ in the VFD module via $\mathcal{R}.\mathsf{recv}()$ and instantiating the predicate $\Psi(\cdot)$ as $\mathsf{Verify}(i||c_i, \sigma_{c_i}, pk_{\mathcal{P}})$. Then $\mathcal{C}$ waits for the execution of VFD to obtain the delivered chunks $((c_1, \sigma_{c_1}), (c_2, \sigma_{c_2}), \cdots)$ and stores them; upon receiving all the n chunks, sends $(\mathsf{delivered})$ to $\mathcal{G}_d$; (iii) waits for the $(\mathsf{revealing})$ message from $\mathcal{G}_d$ to enter the next phase.
- The provider $\mathcal{P}$ would execute as follows: upon receiving $(\mathsf{initiated})$ from $\mathcal{G}_d$, asserts $\Sigma \equiv \mathsf{initiated}$, sends $(\mathsf{mtree}, \mathsf{MT}, \sigma_{\mathcal{P}}^{\mathsf{MT}})$ to $\mathcal{C}$, and enters the next phase.
- The deliverer $\mathcal{D}$ executes as follows: (i) upon receiving $(\mathsf{initiated})$ from $\mathcal{G}_d$, asserts $\Sigma \equiv \mathsf{initiated}$, and then activates the sender $\mathcal{S}$ in the VFD module via $\mathcal{S}.\mathsf{send}()$ and instantiating the predicate $\Psi(\cdot)$ as $\mathsf{Verify}(i||c_i, \sigma_{c_i}, pk_{\mathcal{P}})$, and feeds the VFD module with input $((c_1, \sigma_{c_1}), \ldots, (c_n, \sigma_{c_n}))$; (ii) upon receiving $(\mathsf{getVFDProof})$ from $\mathcal{G}_d$, sends the *latest* receipt, namely $(\mathsf{receipt}, i, \sigma_{\mathcal{C}}^{i})$ to $\mathcal{G}_d$; (iii) waits for the $(\mathsf{revealing})$ message from $\mathcal{G}_d$ to halt.

[6] $\mathcal{P}$ can retrieve the deposits of $\ddot{\mathbb{B}}_{\mathcal{P}}$ and $\ddot{\mathbb{B}}_{\mathsf{pf}}$ back if no deliverers respond timely.
[7] W.l.o.g., we assume $n = 2^k$ for $k \in \mathbb{Z}$ for presentation simplicity.
[8] $\ddot{\mathbb{B}}_{\mathsf{pf}}$ can be required proportional to $(n \times \ddot{\mathbb{B}}_{\mathcal{C}})$ in case $\mathcal{P}$ deliberately lowers it.

Algorithm 1. GenSubKeys algorithm

Input: n, mk
Output: a $(2n-1)$-array KT
let KT be an array of length $(2n-1)$
KT$[0] = \mathcal{H}(mk)$
if $n \equiv 1$ **then**
  return KT
if $n > 1$ **then**
  for i in $[0, n-2]$ **do**
    KT$[2i+1] = \mathcal{H}($KT$[i] \mid\mid 0)$
    KT$[2i+2] = \mathcal{H}($KT$[i] \mid\mid 1)$
return KT

Algorithm 2. RevealKeys algorithm

Input: n, ctr, and mk
Output: rk, an array containing the minimum number of elements in KT that suffices to recover the ctr keys from KT$[n-1]$ to KT$[n+$ ctr $-2]$
let rk and ind be empty arrays
KT $\leftarrow$ GenSubKeys(n, mk)
if ctr $\equiv 1$ **then**
  rk appends $(n-1,$ KT[n $-1])$
  return rk
for i in $[0,$ ctr $-1]$ **do**
  $ind[i] = n-1+i$
while *true* **do**
  let t be an empty array
  for j in $[0, \lfloor |ind|/2 \rfloor - 1]$ **do**
    $p_l = (ind[2j]-1)/2$
    $p_r = (ind[2j+1]-2)/2$
    ▷ merge elements with the same parent node in KT
    if $p_l \equiv p_r$ **then**
      t appends p_l
    else
      t appends $ind[2j]$
      t appends $ind[2j+1]$
  if $|ind|$ is odd **then**
    t appends $ind[|ind|-1]$
  if $|ind| \equiv |t|$ **then**
    break
  $ind = t$
for x in $[0, |ind|-1]$ **do**
  rk appends $(ind[x],$ KT$[ind[x]])$
return rk

Algorithm 3. ValidateRKeys algorithm

Input: n, ctr and erk
Output: *true* or *false* indicating if the correct number (i.e., ctr) of keys can be recovered
if $n \equiv$ ctr and $|erk| \equiv 1$ and the position of $erk[0] \equiv 0$ **then**
  return *true* {root of KT}
Initialize $chunks\_index$ as a set of numbers $\{n-1, \cdots, n+$ ctr $-2\}$
for each $(i, \_)$ in erk **do**
  $d_i = \log(n) - \lfloor \log(i+1) \rfloor$
  $l_i = i$, $r_i = i$
if $d_i \equiv 0$ **then**
  $chunks\_index$ removes i
else
  while $(d_i$--$) > 0$ **do**
    $l_i = 2l_i + 1$
    $r_i = 2r_i + 2$
  $chunks\_index$ removes the elements from l_i to r_i
if $chunks\_index \equiv \emptyset$ **then**
  return *true*
return *false*

Algorithm 4. RecoverKeys algorithm

Input: n, ctr, and rk
Output: a ctr-sized array ks
let ks be an empty array
for each $(i,$ KT$[i])$ in rk **do**
  $n_i = 2^{(\log n - \lfloor \log(i+1) \rfloor)}$
  $v_i =$ GenSubKeys$(n_i,$ KT$[i])$
  ks appends $v_i[n_i - 1 : 2n_i - 2]$
return ks

At the end of this phase, $\mathcal{C}$ receives the encrypted chunks $(c_1, c_2, \dots)$, and $\mathcal{D}$ obtains the payment for the bandwidth contribution of delivering chunks, and the contract updates and records the number of delivered chunks ctr.

Phase III for Reveal. The parties $\mathcal{P}$ and $\mathcal{C}$ interact with $\mathcal{G}_d$ as:

Algorithm 5. ValidatePoM algorithm

Input: $(i, j, c_i, \sigma_{c_i}, \mathcal{H}(m_i), \pi^i_{\mathsf{MT}}, rk, erk, \pi_{\mathsf{VD}})$; also, $(\mathsf{root}_m, n, erk_{\mathsf{hash}}, pk_{\mathcal{P}}, vpk_{\mathcal{C}})$ are stored in the contract and hence accessible
Output: *true* or *false*
1: assert $j \in [0, |erk| - 1]$
2: assert $\mathcal{H}(erk) \equiv erk_{\mathsf{hash}}$
3: assert $\mathsf{VerifyPKE}_{vpk_{\mathcal{C}}}(erk, rk, \pi_{\mathsf{VD}}) \equiv 1$
4: assert $\mathsf{Verify}(i||c_i, \sigma_{c_i}, pk_{\mathcal{P}}) \equiv 1$
5: assert $\mathsf{VerifyMTP}(\mathsf{root}_m, i, \pi^i_{\mathsf{MT}}, \mathcal{H}(m_i)) \equiv 1$
6: $k_i = \mathsf{RecoverChunkKey}(i, j, n, rk)$
7: assert $k_i \neq \perp$
8: $m'_i = \mathsf{SDec}(c_i, k_i)$
9: assert $\mathcal{H}(m'_i) \neq \mathcal{H}(m_i)$
10: **return** *false* in case of any assertion error or *true* otherwise

- The provider $\mathcal{P}$ executes as follows: asserts $\Sigma \equiv \mathsf{revealing}$, executes Algorithm 2 to obtain revealed keys rk, encrypts rk using consumer's $vpk_{\mathcal{C}}$ to obtain erk, and then sends $(\mathsf{revealKeys}, erk)$ to $\mathcal{G}_d$; waits for (sold) from $\mathcal{G}_d$ to halt.
- The consumer $\mathcal{C}$ in this phase would first assert $\Sigma \equiv \mathsf{revealing}$, and wait for $(\mathsf{revealed}, erk)$ from $\mathcal{G}_d$ to execute the following: (i) runs Algorithm 3 to preliminarily check erk, if *false*, sends $(\mathsf{wrongRK})$ to $\mathcal{G}_d$ and halts, or if *true* is returned, decrypts erk to obtain rk via $vsk_{\mathcal{C}}$, and then runs Algorithm 4 to recover chunk keys. Then $\mathcal{C}$ uses these keys to decrypt $(c_1, \cdots, c_{\mathsf{ctr}})$ to obtain $(m'_1, \cdots, m'_{\mathsf{ctr}})$; for each $m'_i, i \in [\mathsf{ctr}]$, checks whether $\mathcal{H}(m'_i)$ is the i-th leaf node in MT received from $\mathcal{P}$ in the *Deliver* phase. If all are consistent, $\mathcal{C}$ outputs $(m'_1, \cdots, m'_{\mathsf{ctr}})$, and then waits for (sold) from $\mathcal{G}_d$ to halt. Otherwise, $\mathcal{C}$ can raise complaint by: choosing one inconsistent position (e.g., the i-th chunk), and then sending $(\mathsf{PoM}, i, j, c_i, \sigma_{c_i}, \mathcal{H}(m_i), \pi^i_{\mathsf{MT}}, rk, erk, \pi_{\mathsf{VD}})$ to $\mathcal{G}_d$, where i is the index of the incorrect chunk to be proved; j is the index of the element in erk that can induce k_i; c_i and σ_{c_i} are the i-th encrypted chunk and its signature received in the *Deliver* phase; $\mathcal{H}(m_i)$ is the value of the i-th leaf node in MT; π^i_{MT} is the Merkle proof for $\mathcal{H}(m_i)$; rk is decryption result from erk; erk is the encrypted revealed key; π_{VD} is the verifiable decryption proof attesting to the correctness of decrypting erk to rk.

Subroutine for Dispute Resolve. For the sake of completeness, the ValidatePoM subroutine of the arbiter contract is detailed in Algorithm 5, which is needed to handle the PoM message sent from consumer. In general, it verifies that the consumer sends the correctly decrypted elements in the key derivation tree KT, and computes the encryption key to a specific chunk (sent from the consumer) according to the KT tree elements, and then it decrypts the chunk and verifies it is essentially committed to the Merkle tree root. Its time complexity corresponds to on-chain cost and is $O(\log n)$, which minimizes the critical on-chain cost.

Analyzing FairDownload. Theorem 1 characterizes the needed security and non-trivial efficiency guarantees of Π_{FD}. The proof is provided in the full version [21].

Theorem 1. *Conditioned on that the underlying cryptographic primitives are secure, the protocol* FairDownload *satisfies the completeness, fairness,*

Algorithm 6. RecoverChunkKey algorithm

Input: (i, j, n, rk)
Output: k_i or $\perp$
$(x, y) \leftarrow rk[j]$
{parse the j-th element in rk to get the key x and the value y}
let k_path be an empty stack
$ind = n + i - 2$ {the index in KT}
if $ind < x$ **then**
  return $\perp$
if $ind \equiv x$ **then**
  return y $\{k_i = y\}$
while $ind > x$ **do**
  k_path pushes 0 if ind is odd
  k_path pushes 1 if ind is even
  $ind = \lfloor (ind - 1)/2 \rfloor$
let $b = |k\_path|$
while $(b\text{--}) > 0$ **do**
  pop k_path to get the value t
  $k_i = \mathcal{H}(y||t)$
return k_i

confidentiality against deliverer, timeliness and non-trivial efficiency properties in the synchronous authenticated network, $\mathcal{G}_d^{\mathsf{ledger}}$-hybrid and stand-alone model.

6 FairStream: Fair p2p Streaming

6.1 FairStream Overview

As shown in Fig. 3, FairStream works as *Prepare*, *Stream*, and *Payout* phases, at a high level. The core ideas are: during streaming, the consumer $\mathcal{C}$ retrieves an encrypted chunk from the deliverer $\mathcal{D}$ and a corresponding decryption key from the provider $\mathcal{P}$, yielding $O(1)$ communication rounds to obtain each chunk; any party may abort in a certain round due to, e.g., untimely response or invalid message; especially, $\mathcal{C}$ can complain during streaming to the arbiter contract $\mathcal{G}_s^{\mathsf{ledger}}$ (abbr. $\mathcal{G}_s$) illustrated in Fig. 4 and get compensated with a valid proof; the *Stream* phase finishes either any party aborts or the timer $\mathcal{T}_{\mathsf{receive}}$ in contract expires; Later, $\mathcal{D}$ and $\mathcal{P}$ can claim the deserved payment by submitting the latest receipt signed by $\mathcal{C}$ before another timer $\mathcal{T}_{\mathsf{finish}}$ (naturally $\mathcal{T}_{\mathsf{finish}} > \mathcal{T}_{\mathsf{receive}}$) in contract expires; the contract determines the ctr, namely the number of delivered chunks or revealed keys, using the *larger* index in $\mathcal{P}$ and $\mathcal{D}$'s receipts. If no receipt is received from $\mathcal{P}$ or $\mathcal{D}$ before $\mathcal{T}_{\mathsf{finish}}$ expires, the index for that party is set as 0. Such a design is critical to ensure fairness as analyzed in the full version [21].

6.2 Π_{FS}: FairStream Protocol

Phase I for Prepare. This phase executes the same as in the Π_{FD} protocol.

Phase II for Stream. The parties $\mathcal{C}$, $\mathcal{D}$ and $\mathcal{P}$ interact with $\mathcal{G}_s$ as follows:

- The consumer $\mathcal{C}$ interested in the content with root_m would initialize a variable $x := 1$ and then: (i) asserts $\Sigma \equiv \mathsf{ready}$, and sends $(\mathsf{consume}, pk_{\mathcal{C}})$ to $\mathcal{G}_s$; (ii) upon receiving $(\mathsf{mtree}, \mathsf{MT}, \sigma_{\mathcal{P}}^{\mathsf{MT}})$ from $\mathcal{P}$, verifies the signature $\sigma_{\mathcal{P}}^{\mathsf{MT}}$ and $\mathsf{root}(\mathsf{MT}) \equiv \mathsf{root}_m$, then stores the Merkle tree MT, or else halts; (iii) on receiving $(\mathsf{deliver}, i, c_i, \sigma_{c_i})$ from $\mathcal{D}$, checks if $i \equiv x \wedge \mathsf{Verify}(i||c_i, \sigma_{c_i}, pk_{\mathcal{P}}) \equiv 1$, if hold, starts (for $i \equiv 1$) a timer $\mathcal{T}_{\mathsf{keyResponse}}$ or resets (for $1 < i \leq n$) it, sends $(\mathsf{keyReq}, i, \sigma_{\mathcal{C}}^i)$ where $\sigma_{\mathcal{C}}^i \leftarrow \mathsf{Sign}(i||pk_{\mathcal{C}}, sk_{\mathcal{C}})$ to $\mathcal{P}$. If failing to check

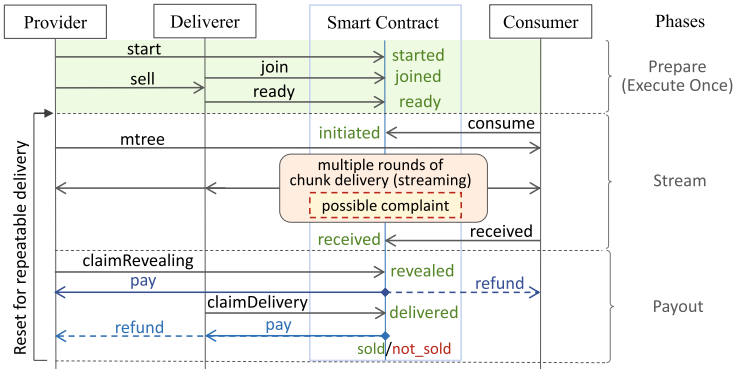


Fig. 3. The overview of FairStream protocol Π_{FS}. The dispute may arise in a certain round in the *Stream* phase, and the messages (claimDelivery) and (claimRevealing) may be sent to the contract in a different order.

or $\mathcal{T}_{\mathsf{keyResponse}}$ times out, halts; (vi) upon receiving $(\mathsf{reveal}, i, k_i, \sigma_{k_i})$ from $\mathcal{P}$ before $\mathcal{T}_{\mathsf{keyResponse}}$ times out, checks whether $i \equiv x \wedge \mathsf{Verify}(i||k_i, \sigma_{k_i}, pk_{\mathcal{P}}) \equiv 1$, if failed, halts. Otherwise, starts to validate the content chunk based on received c_i and k_i: decrypts c_i to obtain m'_i ($m'_i := \mathsf{SDec}_{k_i}(c_i)$) and then checks whether $\mathcal{H}(m'_i)$ is consistent with the i-th leaf node in MT, if inconsistent, sends $(\mathsf{PoM}, i, c_i, \sigma_{c_i}, k_i, \sigma_{k_i}, \mathcal{H}(m_i), \pi^i_{\mathsf{MT}})$ to $\mathcal{G}_s$. If it is consistent, sends the receipts $(\mathsf{receipt}, i, \sigma^i_{\mathcal{CD}})$ to $\mathcal{D}$ and $(\mathsf{receipt}, i, \sigma^i_{\mathcal{CP}})$ to $\mathcal{P}$, where $\sigma^i_{\mathcal{CD}} \leftarrow \mathsf{Sign}(\mathsf{receipt}||i||pk_{\mathcal{C}}||pk_{\mathcal{D}}, sk_{\mathcal{C}})$ and $\sigma^i_{\mathcal{CP}} \leftarrow \mathsf{Sign}(\mathsf{receipt}||i||pk_{\mathcal{C}}||pk_{\mathcal{P}}, sk_{\mathcal{C}})$, and sets $x := x + 1$, and then waits for the next (deliver) message from $\mathcal{D}$. Upon x is set to be $n + 1$, meaning that all chunks are received, sends (received) to $\mathcal{G}_s$; (v) waits for the messages (received) from $\mathcal{G}_s$ to halt.

- The deliverer $\mathcal{D}$ initializes a variable $y := 1$ and executes as: (i) upon receiving $(\mathsf{initiated}, pk_{\mathcal{C}})$ from $\mathcal{G}_s$, sends $(\mathsf{deliver}, i, c_i, \sigma_{c_i}), i = 1$ to $\mathcal{C}$ and starts a timer $\mathcal{T}_{\mathsf{chunkReceipt}}$; (ii) upon receiving $(\mathsf{receipt}, i, \sigma^i_{\mathcal{CD}})$ from $\mathcal{C}$ before $\mathcal{T}_{\mathsf{chunkReceipt}}$ times out, checks whether $\mathsf{Verify}(\mathsf{receipt}||i||pk_{\mathcal{C}}||pk_{\mathcal{D}}, \sigma^i_{\mathcal{CD}}, pk_{\mathcal{C}}) \equiv 1 \wedge i \equiv y$ or not, if succeeds, continues with the next iteration: sets $y := y + 1$, sends $(\mathsf{deliver}, i, c_i, \sigma_{c_i}), i = y$ to $\mathcal{C}$, and resets $\mathcal{T}_{\mathsf{chunkReceipt}}$; If fails or $\mathcal{T}_{\mathsf{chunkReceipt}}$ times out, enters the next phase.
- The provider $\mathcal{P}$ initializes a variable $z := 1$ and executes as: (i) upon receiving $(\mathsf{initiated}, pk_{\mathcal{C}})$ from $\mathcal{G}_s$: sends $(\mathsf{mtree}, \mathsf{MT}, \sigma^{\mathsf{MT}}_{\mathcal{P}})$ to $\mathcal{C}$; (ii) upon receiving $(\mathsf{keyReq}, i, \sigma^i_{\mathcal{C}})$ from $\mathcal{C}$, checks whether $i \equiv z \wedge \mathsf{Verify}(i||pk_{\mathcal{C}}, \sigma^i_{\mathcal{C}}, pk_{\mathcal{C}}) \equiv 1$, if succeeds, sends $(\mathsf{reveal}, i, k_i, \sigma_{k_i})$, where $\sigma_{k_i} \leftarrow \mathsf{Sign}(i||k_i, sk_{\mathcal{P}})$, to $\mathcal{C}$ and starts (for $i \equiv 1$) a timer $\mathcal{T}_{\mathsf{keyReceipt}}$ or resets (for $1 < i \leq n$) it, otherwise enters the next phase; (iii) on receiving $(\mathsf{receipt}, i, \sigma^i_{\mathcal{CP}})$ from $\mathcal{C}$ before $\mathcal{T}_{\mathsf{keyReceipt}}$ expires, checks whether $\mathsf{Verify}(\mathsf{receipt}||i||pk_{\mathcal{C}}||pk_{\mathcal{P}}, \sigma^i_{\mathcal{CP}}, pk_{\mathcal{C}}) \equiv 1 \wedge i \equiv z$, if hold, sets $z := z + 1$, otherwise enters the next phase if $\mathcal{T}_{\mathsf{keyReceipt}}$ times out.

Phase III for Payout. The parties $\mathcal{P}$ and $\mathcal{D}$ interact with $\mathcal{G}_s$ in this phase as:

The Arbiter Contract Functionality $\mathcal{G}_s^{\mathsf{ledger}}$ for p2p Streaming

The contract $\mathcal{G}_s$ can access to ledger, and it interacts with $\mathcal{P}$, $\mathcal{D}$, $\mathcal{C}$ and the adversary $\mathcal{A}$. It locally stores θ, n, root_m, $\mathbb{B}_{\mathcal{P}}$, $\mathbb{B}_{\mathcal{C}}$, $\mathbb{B}_{\mathsf{pf}}$, $\mathsf{ctr}_{\mathcal{D}}$, $\mathsf{ctr}_{\mathcal{P}}$, ctr (all $\mathsf{ctr}_{\mathcal{D}}$, $\mathsf{ctr}_{\mathcal{P}}$, ctr are initialized as 0), $pk_{\mathcal{P}}, pk_{\mathcal{D}}, pk_{\mathcal{C}}$, the penalty flag plt (initialized by $false$), the state Σ and three timers $\mathcal{T}_{\mathsf{round}}$ (implicitly), $\mathcal{T}_{\mathsf{receive}}$, $\mathcal{T}_{\mathsf{finish}}$.

Phase 1: Prepare

- This phase is the same as in $\mathcal{G}_d$.

Phase 2: Stream

- On receive $(\mathsf{consume}, pk_{\mathcal{C}})$ from $\mathcal{C}$:
  - assert $\theta > 0$
  - assert $\mathsf{ledger}[\mathcal{C}] \geq n \cdot \mathbb{B}_{\mathcal{C}} \wedge \Sigma \equiv \mathsf{ready}$
  - store $pk_{\mathcal{C}}$ and let $\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] - n \cdot \mathbb{B}_{\mathcal{C}}$
  - start two timers $\mathcal{T}_{\mathsf{receive}}$, and $\mathcal{T}_{\mathsf{finish}}$
  - let $\Sigma :=$ initiated and send $(\mathsf{initiated}, pk_{\mathcal{C}})$ to all entities
- On receive (received) from $\mathcal{C}$ before $\mathcal{T}_{\mathsf{receive}}$ times out:
  - assert current time $\mathcal{T} < \mathcal{T}_{\mathsf{receive}}$ and $\Sigma \equiv$ initiated
  - let $\Sigma :=$ received and send (received) to all entities
- Upon $\mathcal{T}_{\mathsf{receive}}$ times out:
  - assert current time $\mathcal{T} \geq \mathcal{T}_{\mathsf{receive}}$ and $\Sigma \equiv$ initiated
  - let $\Sigma :=$ received and send (received) to all entities

▷ Dispute resolution in this phase

- On receive $(\mathsf{PoM}, i, c_i, \sigma_{c_i}, k_i, \sigma_{k_i}, \mathcal{H}(m_i), \pi^i_{\mathsf{MT}})$ from $\mathcal{C}$ before $\mathcal{T}_{\mathsf{receive}}$ expires:
  - assert current time $\mathcal{T} < \mathcal{T}_{\mathsf{receive}}$ and $\Sigma \equiv$ initiated
  - assert $\mathsf{Verify}(i||c_i, \sigma_{c_i}, pk_{\mathcal{P}}) \equiv 1$
  - assert $\mathsf{Verify}(i||k_i, \sigma_{k_i}, pk_{\mathcal{P}}) \equiv 1$
  - assert $\mathsf{VerifyMTP}(\mathsf{root}_m, i, \pi^i_{\mathsf{MT}}, \mathcal{H}(m_i)) \equiv 1$
  - $m'_i = \mathsf{SDec}(c_i, k_i)$
  - assert $\mathcal{H}(m'_i) \neq \mathcal{H}(m_i)$
  - let plt $:= true$
  - let $\Sigma :=$ received and send (received) to all entities

Phase 3: Payout

- On receive $(\mathsf{claimDelivery}, i, \sigma^i_{\mathcal{CD}})$ from $\mathcal{D}$:
  - assert current time $\mathcal{T} < \mathcal{T}_{\mathsf{finish}}$
  - assert $i \equiv n$ or $\Sigma \equiv$ received or $\Sigma \equiv$ payingRevealing
  - assert ctr $\equiv 0$ and $0 < i \leq n$
  - assert $\mathsf{Verify}(\mathsf{receipt}||i||pk_{\mathcal{C}}||pk_{\mathcal{D}}, \sigma^i_{\mathcal{CD}}, pk_{\mathcal{C}}) \equiv 1$
  - let $\mathsf{ctr}_{\mathcal{D}} := i$, $\Sigma :=$ payingDelivery, and then send (payingDelivery) to all entities
- On receive $(\mathsf{claimRevealing}, i, \sigma^i_{\mathcal{CP}})$ from $\mathcal{P}$:
  - assert current time $\mathcal{T} < \mathcal{T}_{\mathsf{finish}}$
  - assert $i \equiv n$ or $\Sigma \equiv$ received or $\Sigma \equiv$ payingDelivery
  - assert ctr $\equiv 0$ and $0 < i \leq n$
  - assert $\mathsf{Verify}(\mathsf{receipt}||i||pk_{\mathcal{C}}||pk_{\mathcal{P}}, \sigma^i_{\mathcal{CP}}, pk_{\mathcal{C}}) \equiv 1$
  - let $\mathsf{ctr}_{\mathcal{P}} := i$, $\Sigma :=$ payingRevealing, and then send (payingRevealing) to all entities
- Upon $\mathcal{T}_{\mathsf{finish}}$ times out:
  - assert current time $\mathcal{T} \geq \mathcal{T}_{\mathsf{finish}}$
  - let $\mathsf{ctr} := \max\{\mathsf{ctr}_{\mathcal{D}}, \mathsf{ctr}_{\mathcal{P}}\}$
  - let $\mathsf{ledger}[\mathcal{D}] := \mathsf{ledger}[\mathcal{D}] + \mathsf{ctr} \cdot \mathbb{B}_{\mathcal{P}}$
  - if plt:
    let $\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] + (n - \mathsf{ctr}) \cdot \mathbb{B}_{\mathcal{P}} + \mathsf{ctr} \cdot \mathbb{B}_{\mathcal{C}}$
    let $\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] + (n - \mathsf{ctr}) \cdot \mathbb{B}_{\mathcal{C}} + \mathbb{B}_{\mathsf{pf}}$
  - else:
    let $\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] + (n - \mathsf{ctr}) \cdot \mathbb{B}_{\mathcal{P}} + \mathsf{ctr} \cdot \mathbb{B}_{\mathcal{C}} + \mathbb{B}_{\mathsf{pf}}$
    let $\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] + (n - \mathsf{ctr}) \cdot \mathbb{B}_{\mathcal{C}}$
  - if ctr > 0:
    let $\Sigma :=$ sold and send (sold) to all entities
  - else let $\Sigma :=$ not_sold and send (not_sold) to all entities

▷ Reset to the ready state

- On receive (reset) from $\mathcal{P}$:
  - assert $\Sigma \equiv$ sold or $\Sigma \equiv$ not_sold
  - set ctr, $\mathsf{ctr}_{\mathcal{D}}$, $\mathsf{ctr}_{\mathcal{P}}$, $\mathcal{T}_{\mathsf{receive}}$, $\mathcal{T}_{\mathsf{finish}}$ as 0
  - nullify $pk_{\mathcal{C}}$
  - let $\theta := \theta - 1$ and $\Sigma :=$ ready
  - send (ready) to all entities

Fig. 4. The streaming-setting arbiter functionality $\mathcal{G}_s^{\mathsf{ledger}}$. "Sending to all entities" captures that the smart contract is transparent to the public.

– The provider $\mathcal{P}$ executes as: (i) upon receiving (received) or (delivered) from $\mathcal{G}_s$, or receiving the n-th receipt from $\mathcal{C}$ (i.e., z is set to be $n+1$), sends $(\mathsf{claimRevealing}, i, \sigma^i_{\mathcal{CP}})$ to $\mathcal{G}_s$; (ii) waits for (revealed) from $\mathcal{G}_s$ to halt.
– The deliverer $\mathcal{D}$ executes as: (i) upon receiving (received) or (revealed) from $\mathcal{G}_s$, or receiving the n-th receipt from $\mathcal{C}$ (i.e., y is set to be $n+1$), sends $(\mathsf{claimDelivery}, i, \sigma^i_{\mathcal{CD}})$ to $\mathcal{G}_s$; (ii) waits for (delivered) from $\mathcal{G}_s$ to halt.

Analyzing FairStream**.** Theorem 2 characterizes the security of the protocol Π_{FS}. The proof details are given in the online version [21].

Theorem 2. *Conditioned that the underlying cryptographic primitives are secure, the protocol* FairStream *satisfies the completeness, fairness, confidentiality against deliverer, timeliness, and non-trivial efficiency properties in the synchronous authenticated network,* $\mathcal{G}_s^{\mathsf{ledger}}$*-hybrid and stand-alone model.*

Extension for Delivering from Any Specific Chunk. The protocol FairStream (as well as FairDownload) can be easily tuned to transfer the content from the middle instead of the beginning. Specifically, for the downloading setting, one can simply let the content provider reveal the elements that are able to recover a *sub-tree* of the key derivation tree KT for decrypting the transferred chunks. The complaint of incorrect decryption key follows the same procedure in Sect. 5. For the streaming setting, it is more straightforward as each chunk ciphertext and its decryption key are uniquely identified by the index and can be obtained in $O(1)$ rounds by the consumer, who can immediately complain to contract in the presence of an incorrect decryption result.

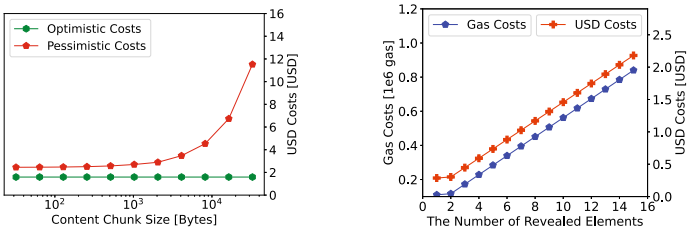
7 Implementation and Evaluations

To shed some light on the feasibility of FairDownload and FairStream, we implement, deploy and evaluate the protocols of Π_{FD} and Π_{FS} in the *Ethereum Ropsten* network. The arbiter contracts are split into *Optimistic* (no dispute) and *Pessimistic* (additionally invoked when dispute occurs) modules. Note that the contracts deployment and the *Prepare* phase can be executed only once and used multiple times to facilitate the delivery of same content to different consumers. Therefore, these one-time costs can be amortized, and we mainly report the costs after the completeness of *Prepare* phase.

In our implementations, the hash function is *keccak256* and the signature is via ECDSA over *secp256k1* curve. The encryption of each chunk m_i with key k_i is instantiated as: parse m_i into t 32-byte blocks $(m_{i,1}, \ldots, m_{i,t})$ and output $c_i = (m_{i,1} \oplus \mathcal{H}(k_i||1), \ldots, m_{i,t} \oplus \mathcal{H}(k_i||t))$. The decryption is same to the encryption. Public key encryption is based on ElGamal: Let $\mathcal{G} = \langle g \rangle$ to be G_1 group over *alt-bn128* curve [36] of prime order q, where g is group generator; the public key $h = g^k$, where $k \leftarrow_{\$} \mathbb{Z}_q$ is the private key; $\mathsf{VEnc}_h(m) = (c_1, c_2) = (g^r, m \cdot g^{kr})$ represents encryption, where the message m is encoded into $\mathcal{G}$ with Koblitz's method [26] and $r \leftarrow_{\$} \mathbb{Z}_q$; the decryption $\mathsf{VDec}_k((c_1, c_2)) = c_2/c_1^k$. To lift ElGamal for verifiable decryption, we adopt Schnorr protocol [37] for Diffie-Hellman tuples with Fiat-Shamir transform [16]. Specifically, $\mathsf{ProvePKE}_k((c_1, c_2))$ works as: run $\mathsf{VDec}_k((c_1, c_2))$ to obtain m; let $x \leftarrow_{\$} \mathbb{Z}_q$, and compute $A = g^x$, $B = c_1^x$, $C = \mathcal{H}(g||A||B||h||c_1||c_2||m)$, $Z = x + kC$, $\pi = (A, B, Z)$, and output (m, π); $\mathsf{VerifyPKE}_h((c_1, c_2), m, \pi)$ works as: parse π to obtain (A, B, Z), compute $C' = \mathcal{H}(g||A||B||h||c_1||c_2||m)$, and verify $(g^Z \equiv A \cdot h^{C'}) \wedge (m^{C'} \cdot c_1^Z \equiv B \cdot c_2^{C'})$, and output 1/0 to indicate whether the verification succeeds or fails.

7.1 Evaluating FairDownload

Table 1 presents the on-chain costs of Π_{FD}, i.e., the costs of all functions in the contract $\mathcal{G}_d$. We adopt a gas price at 10 Gwei to ensure over half of the mining power in Ethereum would mine this transaction,[9] and an exchange rate of 259.4 USD/Ether (the average market price[10] of Ether from 01/01/2020 to 11/03/2020). We stress that utilizing other cryptocurrencies such as Ethereum classic[11] can substantially decrease the actual price of the protocol execution. The price and exchange rate also apply to the streaming setting.



(a) on-chain costs (b) *erk* revealing costs

Fig. 5. Experiment results for the Π_{FD} protocol.

Optimistic Case. Without complaint Π_{FD} executes the functions in *Deliver* and *Reveal* phases, yielding a total cost of 1.032 USD for all involving parties. The on-chain cost is *constant* regardless of the content size and the chunk size, as shown in Fig. 5a. In worse cases, up to $\log(n)$ elements in the key derivation tree KT need to be revealed (where n is the size of content), and the on-chain cost of revealing these elements in KT is depicted in Fig. 5b.

Table 1. Cost of Π_{FD}'s on-chain side $\mathcal{G}_d$

Phase	$\mathcal{G}_d$'s Function	Caller	Gas	USD
Deploy	(Optimistic)	$\mathcal{P}$	2 936 458	7.617
Deploy	(Pessimistic)	$\mathcal{P}$	2 910 652	7.550
Prepare	start	$\mathcal{P}$	110 751	0.287
	join	$\mathcal{D}$	69 031	0.179
	prepared	$\mathcal{D}$	34 867	0.090
Deliver	consume	$\mathcal{C}$	117 357	0.304
	delivered	$\mathcal{C}$	57 935	0.150
	verifyVFDProof	$\mathcal{D}$	56 225	0.146
Reveal	revealKeys	$\mathcal{P}$	113 041	0.293
	payout	$\mathcal{G}_d$	53 822	0.139
Dispute resolve	wrongRK	$\mathcal{C}$	23 441	0.061
	PoM	$\mathcal{C}$	389 050	1.009

Table 2. Cost of Π_{FS}'s on-chain side $\mathcal{G}_s$

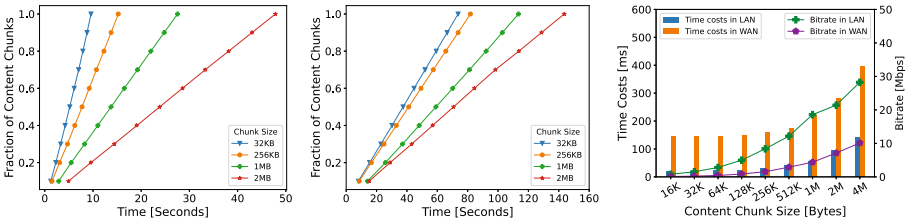
Phase	$\mathcal{G}_s$'s Function	Caller	Gas	USD
Deploy	(Optimistic)	$\mathcal{P}$	1 808 281	4.691
Deploy	(Pessimistic)	$\mathcal{P}$	1 023 414	2.655
Prepare	start	$\mathcal{P}$	131 061	0.340
	join	$\mathcal{D}$	54 131	0.140
	prepared	$\mathcal{D}$	34 935	0.091
Stream	consume	$\mathcal{C}$	95 779	0.248
	received	$\mathcal{C}$	39 857	0.103
	receiveTimeout	$\mathcal{G}_s$	39 839	0.103
	PoM	$\mathcal{C}$	90 018	0.234
Payout	claimDelivery	$\mathcal{D}$	67 910	0.176
	claimRevealing	$\mathcal{P}$	67 909	0.176
	finishTimeout	$\mathcal{G}_s$	88 599	0.230

[9] https://ethgasstation.info/.
[10] https://www.coindesk.com/price/ethereum/.
[11] https://ethereumclassic.org/.

Pessimistic Case. When some complaint arises by the consumer, the arbiter contract involves to resolve it. The cost of executing wrongRK function relates to the concrete values of n, ctr and $|erk|$, and in Table 2, the cost is evaluated on $n \equiv \mathsf{ctr} \equiv 512$, and $|erk| \equiv 1$. The cost of PoM function validating misbehavior varies by the content chunk size η, as depicted in Fig. 5a pessimistic costs. The results demonstrate that the on-chain costs increase linearly in the chunk size.

7.2 Evaluating FairStream

Table 2 lists the costs of contract $\mathcal{G}_s$ to show the on-chain expense of the streaming protocol Π_{FS}.



(a) Time cost of streaming 512 content chunks in LAN

(b) Time cost of streaming 512 content chunks in WAN

(c) Average time cost and the corresponding bitrate

Fig. 6. The performance of FairStream protocol in the LAN and WAN.

Optimistic Case. Without dispute, Π_{FS} executes the functions in *Stream* and *Payout* phases except the PoM function for verifying proof of misbehavior, yielding a total cost of 0.933 USD for all involved parties. The costs in this mode is *constant* regardless of the content size and chunk size.

Pessimistic Case. With complaint, the total on-chain cost is 1.167 USD for all involved parties. The cost of the PoM function: (i) increases slightly in number of chunks n, since it computes $O(\log n)$ hashes to verify the Merkle proof; (ii) increase linearly in the chunk size η for the chunk decryption, which follows a similar trend to Fig. 5a pessimistic costs but with lower costs as no verification of verifiable decryption proof is needed.

Streaming Efficiency. Fig. 6a and b illustrate the efficiency of streaming 512 chunks (via Π_{FS}) of various sizes in both LAN and WAN.[12] Results indicate that: (i) obviously the time costs increase due to the growth of chunk size; (ii) the delivery process remains stable with only slight fluctuation, as reflected by the slope for each chunk size in Fig. 6a and b. Furthermore, Fig. 6c depicts the average latency for delivering each chunk (over the 512 chunks) and the corresponding bitrate. It shows that the bitrate can reach 10 Mbps even in the public network, which is potentially sufficient to support high-quality content streaming. E.g., the video bitrate for HD 1080 are *at most* 8 Mbps [22].

[12] The experimental configuration details are described in the full version [21].

8 Conclusion

We present the first two fair p2p content delivery protocols atop blockchains to support *fair p2p downloading* and *fair p2p streaming*, respectively. They enjoy strong fairness guarantees to protect any of the content provider, the content consumer, and the content deliverer from being ripped off by other colluding parties. Essentially, our new delivery fairness notion circumvents the limit of conventional exchange fairness, thus supporting to pay the delivers according to their bandwidth usages. Detailed complexity analysis and extensive experiments are conducted to demonstrate the high efficiency of our designs. Yet still, the area is largely unexplored and has a few immediate follow-ups, for example: (i) in order to realize maximized delivery performance, it is desirable to design a mechanism of adaptively choosing deliverers during each delivery task; (ii) it is also enticing to leverage the off-chain payment channels to handle possible micropayments and further reduce the on-chain cost.

Acknowledgements. Yuan Lu is supported in part by the National Natural Science Foundation of China (No. 62102404). This work was also supported in part by NSF #1801492 and FHWA 693JJ320C000021. The authors would like to thank the anonymous reviewer for their valuable comments and suggestions.

References

1. Akamai: Akamai (2021). https://www.akamai.com/
2. Almashaqbeh, G.: CacheCash: a cryptocurrency-based decentralized content delivery network. Ph.D. thesis, Columbia University (2019)
3. Anjum, N., Karamshuk, D., Shikh-Bahaei, M., Sastry, N.: Survey on peer-assisted content delivery networks. Comput. Netw. **116**, 79–95 (2017)
4. Asokan, N., Shoup, V., Waidner, M.: Optimistic fair exchange of digital signatures. IEEE J. Sel. Areas Commun. **18**, 593–610 (2000)
5. Benet, J.: IPFS-content addressed, versioned, p2p file system. arXiv preprint arXiv:1407.3561 (2014)
6. Bentov, I., Kumaresan, R.: How to use bitcoin to design fair protocols. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. LNCS, vol. 8617, pp. 421–439. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44381-1_24
7. Blum, M.: How to exchange (secret) keys. In: Proceedings of the ACM STOC 1983, pp. 440–447 (1983)
8. Camenisch, J., Shoup, V.: Practical verifiable encryption and decryption of discrete logarithms. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 126–144. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45146-4_8
9. Choudhuri, A.R., Green, M., Jain, A., Kaptchuk, G., Miers, I.: Fairness in an unfair world: fair multiparty computation from public bulletin boards. In: Proceedings of the ACM CCS, pp. 719–728 (2017)
10. Damgård, I.B.: Practical and provably secure release of a secret and exchange of signatures. J. Cryptol. **8**, 201–222 (1995). https://doi.org/10.1007/BF00191356
11. Dziembowski, S., Eckey, L., Faust, S., Malinowski, D.: Perun: virtual payment hubs over cryptocurrencies. In: Proceedings of the IEEE S&P 2019, pp. 106–123 (2019)

12. Dziembowski, S., Eckey, L., Faust, S.: FairSwap: how to fairly exchange digital goods. In: Proceedings of the ACM CCS, pp. 967–984 (2018)
13. ENISA: ENISA threat landscape 2020 -botnet (2020). https://www.enisa.europa.eu/publications/enisa-threat-landscape-2020-botnet
14. Fan, B., Lui, J.C., Chiu, D.M.: The design trade-offs of BitTorrent-like file sharing protocols. IEEE/ACM Trans. Network. **17**, 365–376 (2008)
15. Feldman, M., Lai, K., Stoica, I., Chuang, J.: Robust incentive techniques for peer-to-peer networks. In: Proceedings of the ACM EC (2004)
16. Fiat, A., Shamir, A.: How to prove yourself: practical solutions to identification and signature problems. In: Odlyzko, A.M. (ed.) CRYPTO 1986. LNCS, vol. 263, pp. 186–194. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-47721-7_12
17. Filecoin: Filecoin: a decentralized storage network. https://filecoin.io/filecoin.pdf
18. Filecoin: Filecoin spec. (2020). https://spec.filecoin.io/
19. Goyal, P., Netravali, R., Alizadeh, M., Balakrishnan, H.: Secure incentivization for decentralized content delivery. In: 2nd USENIX Workshop on Hot Topics in Edge Computing (2019)
20. Hardin, G.: The tragedy of the commons. J. Nat. Resour. Policy Res. **1**, 243–253 (2009)
21. He, S., Lu, Y., Tang, Q., Wang, G., Wu, C.Q.: Fair peer-to-peer content delivery via blockchain. arXiv preprint arXiv:2102.04685 (2021)
22. IBM: Internet connection and recommended encoding settings. https://support.video.ibm.com/hc/en-us/articles/207852117-Internet-connection-and-recommended-encoding-settings
23. Janin, S., Qin, K., Mamageishvili, A., Gervais, A.: FileBounty: fair data exchange. In: IEEE (EuroS&PW), pp. 357–366 (2020)
24. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. CRC Press (2014)
25. Kiayias, A., Zhou, H.-S., Zikas, V.: Fair and robust multi-party computation using a global transaction ledger. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9666, pp. 705–734. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49896-5_25
26. Koblitz, N.: Elliptic Curve Cryptosystems. Mathematics of Computation, pp. 203–209 (1987)
27. Kosba, A., Miller, A., Shi, E., Wen, Z., Papamanthou, C.: Hawk: the blockchain model of cryptography and privacy-preserving smart contracts. In: Proceedings of the IEEE (S&P), pp. 839–858 (2016)
28. Küpçü, A., Lysyanskaya, A.: Usable optimistic fair exchange. In: Pieprzyk, J. (ed.) CT-RSA 2010. LNCS, vol. 5985, pp. 252–267. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11925-5_18
29. Levin, D., LaCurts, K., Spring, N., Bhattacharjee, B.: Bittorrent is an auction: analyzing and improving bittorrent's incentives. ACM SIGCOMM Comput. Commun. Rev. **38**, 243–254 (2008)
30. Locher, T., Moore, P., Schmid, S., Wattenhofer, R.: Free riding in bittorrent is cheap. In: HotNets (2006)
31. Maxwell, G.: The first successful zero-knowledge contingent payment (2016). https://bitcoincore.org/en/2016/02/26/zero-knowledge-contingent-payments-announcement/
32. Mehar, M.I., et al.: Understanding a revolutionary and flawed grand experiment in blockchain: the DAO attack. J. Cases Inf. Technol. **21**, 19–32 (2019)

33. Miller, A., Bentov, I., Bakshi, S., Kumaresan, R., McCorry, P.: Sprites and state channels: payment networks that go faster than lightning. In: Goldberg, I., Moore, T. (eds.) FC 2019. LNCS, vol. 11598, pp. 508–526. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32101-7_30
34. Miller, A., Juels, A., Shi, E., Parno, B., Katz, J.: Permacoin: repurposing bitcoin work for data preservation. In: Proceedings of the IEEE S&P, pp. 475–490 (2014)
35. Piatek, M., Isdal, T., Anderson, T., Krishnamurthy, A., Venkataramani, A.: Do incentives build robustness in bittorrent. In: Proceedings of the NSDI (2007)
36. Reitwiessner, C.: EIP-196: precompiled contracts for addition and scalar multiplication on the elliptic curve alt_bn128. Ethereum Improvement Proposals, No. 196 (2017). https://eips.ethereum.org/EIPS/eip-196
37. Schnorr, C.P.: Efficient identification and signatures for smart cards. In: Quisquater, J.-J., Vandewalle, J. (eds.) EUROCRYPT 1989. LNCS, vol. 434, pp. 688–689. Springer, Heidelberg (1990). https://doi.org/10.1007/3-540-46885-4_68
38. Sherman, A., Nieh, J., Stein, C.: FairTorrent: a deficit-based distributed algorithm to ensure fairness in peer-to-peer systems. IEEE/ACM TON **20**, 1361–1374 (2012)
39. Shin, K., Joe-Wong, C., Ha, S., Yi, Y., Rhee, I., Reeves, D.S.: T-chain: a general incentive scheme for cooperative computing. IEEE/ACM TON **25**, 2122–2137 (2017)
40. Sirivianos, M., Park, J.H., Yang, X., Jarecki, S.: Dandelion: cooperative content distribution with robust incentives. In: Proceedings of the USENIX ATC (2007)
41. StorJ: Storj (2018). https://storj.io/storj.pdf
42. Swarm (2020). https://swarm.ethereum.org/
43. Wood, G., et al.: Ethereum: a secure decentralised generalised transaction ledger. Ethereum project yellow paper, pp. 1–32 (2014)