# ConTrib

## Maintaining fairness in decentralized big tech alternatives by accounting work

de Vos, Martijn; Pouwelse, Johan

**DOI**
10.1016/j.comnet.2021.108081

**Publication date**
2021

**Document Version**
Final published version

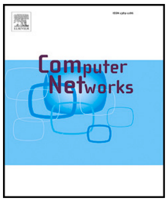
**Published in**
Computer Networks

**Citation (APA)**
de Vos, M., & Pouwelse, J. (2021). ConTrib: Maintaining fairness in decentralized big tech alternatives by accounting work. *Computer Networks*, *192*, Article 108081. https://doi.org/10.1016/j.comnet.2021.108081

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet



# ConTrib: Maintaining fairness in decentralized big tech alternatives by accounting work

Martijn de Vos [*], Johan Pouwelse

*Delft University of Technology, Netherlands*

ARTICLE INFO

*Keywords:*
Fairness
Decentralized applications
Accounting mechanism
Free-rider prevention

ABSTRACT

“Big Tech” companies provide digital services used by billions of people. Recent developments, however, have shown that these companies often abuse their unprecedented market dominance for selfish interests. Meanwhile, decentralized applications without central authority are gaining traction. Decentralized applications critically depend on its users working together. Ensuring that users do not consume too many resources without reciprocating is a crucial requirement for the sustainability of such applications.

We present ConTrib, a universal mechanism to maintain fairness in decentralized applications by accounting the work performed by peers. In ConTrib, participants maintain a personal ledger with tamper-evident records. A record describes some work performed by a peer and links to other records. Fraud in ConTrib occurs when a peer illegitimately modifies one of the records in its personal ledger. This is detected through the continuous exchange of random records between peers and by verifying the consistency of incoming records against known ones. Our simple fraud detection algorithm is highly scalable, tolerates significant packet loss, and exhibits relatively low fraud detection times. We experimentally show that fraud is detected within seconds and with low bandwidth requirements. To demonstrate the applicability of our work, we deploy ConTrib in the Tribler file-sharing application and successfully address free-riding behaviour. This two-year trial has resulted in over 160 million records, created by more than 94’000 users.

## 1. Introduction

Over the last decades, “Big Tech” companies have obtained an unprecedented market dominance in the industry for information technology [1]. Companies such as Google, Amazon, Facebook, and Apple are omnipresent in our current society and even have the means of acting as small states, inhabited by billions of users worldwide. By continuously broadening their activities, these companies seek to expand their virtual territory and seek to obtain monopolistic control over the enabling elements for digital services, such as access to the Internet [2].

The societal impact of “Big Tech” companies is a double-edged sword. On the one hand, these companies are facilitating new modes of digital interaction between users and enable new business models. The sharing economy is a prime example of this phenomena. It is made up by digital markets for the trustworthy exchange of personal assets (e.g., houses and cars) between strangers [3]. Sharing personal assets is a concept that has long been confined to trusted individuals, such as family and friends [4]. Likewise, media platforms such as YouTube provide the required infrastructure for new forms of user engagement through video weblogging or “vlogging”.

On the other hand, it has become apparent that “Big Tech” companies tend to exploit their established market position and are increasingly involved in regulatory or political battles. This behaviour sometimes goes undetected for years. For example, researchers have only recently demonstrated that Uber actively manipulates the matchmaking process between passengers and drivers for commercial interests, therefore decreasing platform fairness and income equality of drivers [5]. Similarly, Apple is currently under antitrust investigation by the European Commission that is assessing whether Apples’ rules for developers on the distribution of apps via the App Store violate competition rules [6].

These concerning developments have contributed to an increase in the deployment of *decentralized* applications. Decentralized applications avoid centralized ownership and delegate the decision making away from a single authority. A decentralized application mainly operates through the direct cooperation and information exchange between users, which we call *peers*. Arguably, Bitcoin is the most influential solution in this direction and provides a decentralized cash system without the supervision by an authoritative bank [7]. The underlying data structure of Bitcoin, a blockchain, is at the core of numerous decentralized

[*] Corresponding author.
*E-mail address:* m.a.devos-1@tudelft.nl (M. de Vos).

https://doi.org/10.1016/j.comnet.2021.108081
Received 20 November 2020; Received in revised form 23 March 2021; Accepted 29 March 2021
Available online 5 April 2021

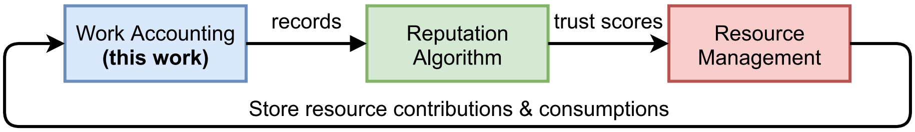


**Fig. 1.** Addressing fairness issues in decentralized networks through work accounting, reputation and resource allocation. This work introduces a lightweight mechanism for secure work accounting.

applications [8]. At the time of writing, there are thousands of decentralized applications deployed on the Ethereum blockchain alone [9]. These decentralized applications include marketplaces, auctions, voting systems, lotteries, and games.

In contrast to the applications deployed by "Big Tech" companies, decentralized applications are fully maintained by peers, without coordination by a third party. Decentralized applications require peers to pool their computer resources to provide the desired services to participants. Specifically, peers have to communicate with other peers, have to dedicate computational power to process incoming network messages, and frequently have to store data generated by other peers. Some decentralized applications critically depend on the voluntary contribution of computer resources by peers. Bitcoin, for example, prevents the uncontrolled minting of digital coins through a resource-based consensus mechanism executed by miners [7]. These miners continuously attempt to solve a computational puzzle, a resource-intensive task that decides who can append transactions to the blockchain ledger. Another volunteer-based application is Tor, providing anonymity by routing Internet traffic through user-operated relay and exit nodes [10].

Unfortunately, long-term cooperation between peers in decentralized applications is non-trivial to achieve. Not rewarding peers for performing work can result in an unfair situation where peers enjoy the services provided by others, without contributing computer resources in return. This detrimental behaviour, also called *free-riding*, can degrade network health in the long term, as dedicated peers will ultimately leave [11]. Measurements have shown that free-riding often prevails in cooperative applications such as BitTorrent and Tor [12]. Since the cooperation between peers is at the heart of decentralized technology, we argue that this form of fairness is a crucial requirement for *any* decentralized application to ensure long-term sustainability [13]. With the renewed interest in decentralized alternatives for "Big Tech", ensuring fairness in decentralized applications is a significant challenge.

A promising approach to address these fairness issues is by deploying a decentralized reputation mechanism, and allocate resource based on trust scores of individuals. This process is visualized in Fig. 1. First, users account all performed and consumed work in the network within records. A reputation mechanism then computes trustworthiness scores of users, based on created records. A user decides who to help based on a resource management algorithm. In general, users with low reputation scores should be refused services whereas trusted users enjoy preferential treatment from others. There currently is no accounting mechanism that is specifically built to account work performed and consumed by peers in decentralized networks, to the best of our knowledge.

**Our solution.** We specifically focus on the accounting of work performed by peers, which is crucial to ensure fairness within decentralized applications. In this work, we design, implement and evaluate a universal data store, named ConTrib. ConTrib is capable of accounting work within decentralized applications that rely on the work performed by participating peers. Examples of work include storing files on behalf of other peers, performing computations, or relaying network packets. With ConTrib, each peer maintains a *personal ledger* with tamper-evident *records*. The ConTrib records can then be used by an application to determine the trustworthiness of individuals, e.g., with a reputation algorithm. Consequently, users have a natural incentive to increase their social standing by modifying or removing records. This misbehaviour is a key threat to the integrity of the ConTrib data structure.

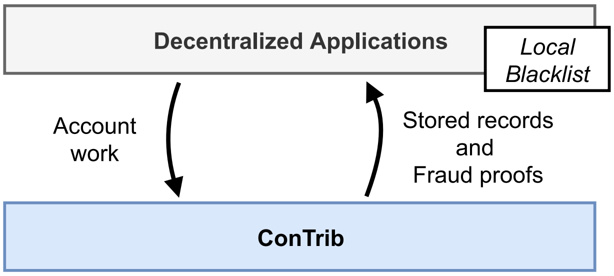


**Fig. 2.** Decentralized applications can use ConTrib to account the work performed by peers within tamper-evident *records*. These records are used by connected applications to detect free-riders and fraudsters, which are added to a local blacklist. Applications can then choose to refuse services to the peers on the blacklist.

We refer to the illegitimate modification of a record as fraud. To detect fraud, peers continuously request random records from other peers and disseminate newly created records in the network. Peers verify the consistency of incoming records with the ones stored in their database.

ConTrib enables connected applications to select which work should be accounted. Fig. 2 shows how a decentralized application can leverage ConTrib to account work. By inspecting the records in personal ledgers, an application can gather evidence of free-riding behaviour. Each application maintains a local blacklist with both free-riders and peers that have committed fraud. Peers refrain from performing work for peers on the blacklist. ConTrib can be deployed to alleviate fairness concerns in work-based decentralized applications, which include peer-assisted video distribution, anonymous communication networks, and distributed learning environments.

We implement ConTrib and evaluate how different parameters impact the efficiency of fraud detection and the network usage. We find that fraud can be detected within seconds on average, even in larger networks with 10'000 peers where every peer commits fraud, and under a conservative strategy for record exchange. We also show that ConTrib highly tolerates packet loss.

To show the effectiveness of ConTrib in a realistic environment, we employ our accounting mechanism to address free-riding behaviour in Tribler. Tribler is a decentralized application downloaded by over 1.7 million users [14]. We specifically use ConTrib to account bandwidth exchanges in Triblers' Tor-like overlay and use the accounted work to refuse services to free-riders. Our two-year measurements have resulted in over 160 million records, created by more than 94'000 users. This large-scale deployment trial is a key milestone in our ongoing research effort to solve the tragedy-of-the-commons within Internet communities [15].

The main contribution of this work is four-fold:

1. ConTrib, a *universal mechanism* that maintains fairness in decentralized applications by accounting work (Section 3).
2. An efficient *fraud detection mechanism* to detect the illegitimate tampering of created records in ConTrib (Section 4).
3. An *implementation* and *evaluation* of ConTrib with up to 10'000 peers, demonstrating the scalability of our mechanism and showing that fraud can be detected within seconds on average (Sections 5 and 6).
4. A *two-year deployment trial* of ConTrib in Tribler, involving 94'000 Internet-recruited volunteers. This trial successfully addresses free-riding behaviour in Tribler (Section 7).

## 2. Background and problem description

This work addresses fairness issues in decentralized applications, in particularly free-riding behaviour. Many decentralized applications integrate a mechanism to reward peers for performing work [16]. We first outline two incentive mechanisms that address free-riding by peers, namely trade-based and trust-based incentives [17].

### 2.1. Trade-based incentives

With trade-based incentives, performed work by peers is remunerated using a credit or payment system. Peers that use the services of other peers are required to pay for that service. Remuneration either occurs immediately after the work is performed or when a certain number of payments is outstanding. The accrued credits can either be converted to real-world money or are merely useful to show the dedication of a particular peer. BOINC is a well-known volunteer computing project that rewards users with virtual credits for processing scientific workloads [18].

Blockchain technology also relies on financial remuneration to keep the system secure [19]. Miners, dedicated peers that maintain the blockchain ledger, are usually rewarded for their efforts. Specifically, users pay a small fee for each transaction and miners then these fees when including their transactions in the blockchain. Other decentralized applications have adopted cryptocurrencies as a payment system to reward the performed work. Filecoin is a decentralized system where users pay with a blockchain-based token to have their data stored by peers [20]. Likewise, TorCoin proposes a mechanism where the relay and exit nodes “mine” a Bitcoin-derived cryptocurrency by relaying Internet traffic [21].

Even though trade-based incentives are frequently used to incentivize work, remuneration is not an adequate solution for any decentralized applications, for the following three reasons [22]. First, they require the integration of a secure payment infrastructure which complicates the system design and potentially enables new forms of attack, such as coin forgery and double-spending. Using a central authority to keep track of each peer’s balance introduces a central component and poses a single-point-of-failure. Second, remuneration requires peers to determine the price of a digital service, which can be hard to estimate. Third, remuneration can result in new forms of unfairness where a few affluent peers exclusively enjoy the services of a decentralized application. This situation could arise when operating peer-to-peer auctions for the allocation of services.

### 2.2. Trust-based incentives

Applications implementing trust-based incentives indirectly reward community members for their work. For example, the system can reward dedicated peers with preferential treatment or provide them access to exclusive services. This approach often requires peers to keep track of the long-term contributions of other peers using accounting infrastructure [23]. The specifications of accounted work can then be used by the application to detect how a particular peer has contributed to the system. For instance, the accounted work can be used by a reputation algorithm that outputs a ranking of peers [24]. If the ranking of a specific peer is below a threshold, the application can decide to refuse to perform work for this peer until its ranking has improved. We outline related work that uses work accounting and trust-based incentives. For an overview of (decentralized) reputation mechanisms and trust models, we refer the interested reader to existing work [25,26].

Perhaps the most popular decentralized application is BitTorrent, a peer-to-peer file exchange protocol [27]. In BitTorrent, each peer has a limited number of slots to allocate to other peers. The system uses tit-for-tat, a cooperation strategy where a counterparty loses its slot when it stops to reciprocate. This simple strategy leads to higher network utilization since long-term free-riders will not be allocated slots. BitTorrent does not persist all contributions and consumptions of other peers, of but tracks the performance of connected peers for each download.

The InterPlanetary File System (IPFS) is a decentralized system for file storage and exchange [28]. IPFS breaks up files into blocks, which are identifiable by a content identifier. The original IPFS whitepaper describes bitSwap, a set of tools to exchange blocks while addressing free-riding behaviour through block bartering. It ensures that peers are incentivized to seed blocks by pair-wise tracking of outstanding “balances”. Peers that do not sufficiently share blocks will be ignored by others.

Wallach et al. present different mechanisms for the fair sharing of resources in decentralized applications [29]. These mechanisms ensure that each peer maintains a log with actions and includes random auditing of logs. The applicability of their work is exclusive to storage-based application and is not reusable for other decentralized applications. Osipkov et al. describe an accounting mechanism for file-sharing applications [30]. Specifically, each peer maintains a set of witnesses that monitors all transactions of that peer.

LiFTinG and AcTinG are protocols for tracking free-riding behaviour in gossip-based applications [31,32]. The LiFTinG protocol exploits the message dynamics between peers and verifies that the content received by a peer is further propagated according to the protocol. The design depends on a statistical approach and cross-checking of logs to detect free-riders but is not reusable for applications beyond gossip. AcTinG is a gossip-based dissemination protocol that is resistant against colluding rational peers.

Other approaches maintain a distributed ledger that store information in decentralized applications. Seuken and Parkes introduce a Sybil-resistant accounting mechanism based on transitive trust [33]. PeerReview is an accountability mechanism to record message exchange between peers [34]. Peers store all network messages in a local log. Dedicated witnesses continuously audit peers and detect whether a peer has deviated from the protocol. The FullReview protocol extends PeerReview by addressing selfish behaviour with a game-theoretical model [35]. Otte et al. present TrustChain, a Sybil-resistant reputation mechanism with an accompanying accounting mechanism [36]. The authors apply their mechanism to address free-riding behaviour in a file-sharing network. We find that peers in TrustChain cannot engage in the recording of multiple interactions simultaneously, significantly limiting the achievable throughput. Crosby et al. present a data structure for tamper-evident logging [37]. This data structure orients around the efficient logging of unilateral system events on a server. Peermint is an accounting mechanism designed for market-based management of decentralized applications [38].

### 2.3. Problem description

There currently is no *universal* accounting mechanism that can be used to address fairness issues in decentralized applications, to the best of our knowledge. We address this shortcoming and describe three challenges when designing such a mechanism.

**Challenge I: Universality.** The trust-based solutions that we have identified so far are designed for usage within a single application domain and are infeasible to re-use. We believe that universality is an important property to address fairness concerns in novel decentralized applications.

**Challenge II: Full decentralization without central authority.** To keep our system reusable and universal, we avoid *any* decision making by entities with leveraged authorities and central servers. The lack of a central authority makes our mechanism *fully decentralized* and easier to deploy. In general, decentralized mechanisms are less vulnerable to large-scale attacks, tend to scale better, and are more resilient to failure. They also are an excellent architectural fit with existing decentralized applications without central authorities.

**Challenge III: Fraud detection.** Peers have a natural incentive to misrepresent the magnitude of their efforts to inflate their social standing or to hide information unfavourable to their standing [23]. Our accounting mechanism must address the completeness and correctness of the stored information. We must *detect the manipulation or hiding of accounted information* and punish adversarial peers accordingly. We consider the accounting of events that have not occurred in the application out of scope.

## 3. Accounting work with ConTrib

The design of our universal accounting mechanism, named *ConTrib*, is inspired by the tamper-evident properties of blockchain but does not require peers to reach consensus on a coherent history of records. Instead, ConTrib optimistically detects the illegitimate modification of records while keeping the computational overhead and bandwidth requirements low. Decentralized applications can account the work performed by peers within tamper-evident *records*. A record describes some work performed by one peer for another peer. Each peer organizes their records in a *personal ledger*. Records point to prior records in the same personal ledger and also point to records in the personal ledger of others. The latter pointer captures an agreement between two peers. Peers continuously exchange records with other random peers and request records in the personal ledgers of others. By validating the consistency of incoming records against known ones, a peer can irrefutably prove fraud attempts to other peers.

We further elaborate on the design of ConTrib. We first outline the network and threat model. We then describe the ConTrib data structure and show how ConTrib accounts the work in decentralized applications.

### 3.1. Network model

The ConTrib mechanism is built on a peer-to-peer network. We assume an unstructured network structure. Unstructured networks are relatively straightforward to maintain and are highly resilient against churn. We assume that the used networking library handles network bootstrapping and peer discovery. We also assume that the communication channels between peers are unreliable and unordered (e.g., by using the UDP communication model). The arrival time of messages is not upper-bounded, and outbound messages can fail to arrive at their intended destination. Each peer has a cryptographic key pair, consisting of a public and private key. The public key acts as a unique identifier of the peer in the network, whereas the private key is used to sign records and outgoing network messages. We consider attacks targeted at the network layer, e.g., the Eclipse Attack, outside the scope of this work.

A significant threat in Internet-deployed applications is the Sybil Attack, where an adversary operates multiple identities to subvert the network [39]. The Sybil Attack frequently occurs in open Internet communities, where the cost of creating a new digital identity is often negligible. Although the ConTrib mechanism does not include defences against Sybil identities, we argue that this threat can be mitigated with well-established techniques that complement ConTrib in a deployment setting. A basic defence mechanism is to have peers solve a computational puzzle when they wish to join the network [40]. In addition, using a Sybil-resistant reputation mechanism that processes ConTrib records can effectively mitigate the effect of Sybil identities on computed trust scores [41,42]. We also consider self-sovereign identities as a promising solution that can bolster decentralized networks with long-term identities [43].

We leave defences against misreporting, the accounting of work that has not occurred in the application, to other layers in the application stack. This attack is closely related to the Sybil Attack since Sybil identities are likely to create fake interactions amongst them [44]. Misreporting is challenging to address in a generic manner since there is not always a straightforward method to assess if some accounted work is legitimate. Some protocols use cryptographic techniques to prove the accuracy of performed work, for example, Proof-of-Storage and Proof-of-Bandwidth [20,21]. These proofs, however, cannot easily be used across different application domains.

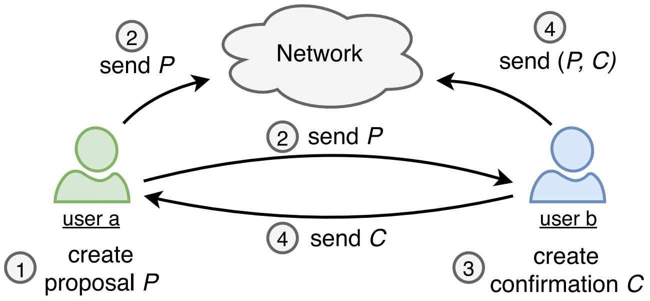


**Fig. 3.** The process of recording work between peers $a$ and $b$ within two records: a proposal $P$ and a confirmation $C$.

### 3.2. Threat model

Our threat model orients around malicious peers that attack the integrity of the ConTrib data structure. This attack proceeds through the strategic modification of ConTrib records. For example, a peer can inflate the amount of work it has performed by modifying one of the records in its personal ledger. We refer to the illegitimate modification of ConTrib records as *fraud*. Even though this definition may seem limited, we argue that this kind of fraud is a fundamental threat to the ConTrib data structure. In particular, our definition of fraud also entails a more advanced form of record manipulations where peers collude to erase a particular interaction from history.[1] In a reputation system, for example, this would happen when a well-trusted peer temporarily boosts the reputation of another peer by accounting some work and then attempts to hide the existence of this interaction later. Reverting this interaction requires both counterparties to either override or remove the associated records, which we consider as fraud. We note that a particular fraud instance in our system involves at most two guilty peers. As we discussed in Section 2.3, we require that fraud is detected. We assume that the computing power of adversaries is bounded and that cryptographic primitives are secure.

### 3.3. Recording interactions

Some work that involves peers $a$ and $b$ is recorded using two records: one *proposal* created by $a$ and one *confirmation* created by $b$. W.l.o.g., we assume that the accounted work is performed by peer $a$ for peer $b$. The process of accounting this work is visualized in Fig. 3. First, $a$ creates a proposal record, which we refer to as $P$ (step ①). Proposal $P$, created by peer $a$, is a tuple with the following four attributes:

$$P = (\texttt{pubKey}, \texttt{pubKeyOther}, \texttt{payload}, \texttt{sig})$$

$P$ contains the public key of peers $a$ and $b$ (`pubKey` and `pubKeyOther`, respectively), an application-specific payload (`payload`), and a digital signature (`sig`) created by $a$ of the record in binary form. The `payload` attribute is an arbitrary blob of data and is provided by the connected application. To increase the resilience against manipulation, we extend records with additional fields in the next section. After peer $a$ has included all described attributes in the proposal, it persists the record to its database, sends the proposal to $b$, and disseminated the proposal to $f$ random peers in the network (step ②). We refer to $f$ as the *fanout* value.

[1] Peers might refrain from overriding or erasing their records during or after a collusion attempt. We do not consider this as fraud since adversaries do not exploit the ConTrib data structure. Since all records associated with the collusion attempt are accounted, decentralized applications might employ additional logic to analyse created records and attempt to detect possible collusion attempts, e.g., with correlation analysis [45].

When peer $b$ receives the proposal $P$, $b$ verifies its validity. It is during this step that fraud is detected. The validation logic of incoming records is elaborately discussed in Section 4. If the incoming proposal $P$ is deemed valid, the connected application determines if the payload in $P$ truthfully describes the performed work. If $P$ considered invalid, $b$ ignores the incoming proposal and takes no further action. Otherwise, $b$ creates a confirming record that confirms $P$ (step ③). This confirmation, denoted by $C$, contains the same attributes as the proposal $P$ and also includes the hash of $P$. Confirmation $C$, created by peer $b$ is a tuple with the following five attributes:

$$C = (\texttt{pubKey}, \texttt{pubKeyOther}, \texttt{payload}, \texttt{proposalHash}, \texttt{sig})$$

The value of `proposalHash` is computed by $H(P)$, where $H(\cdot)$ is a secure hash function. We call the `proposalHash` attribute in $C$ the *confirmation pointer*. After the creation of $C$, peer $b$ persists the confirmation to its database, sends it to $a$, and disseminates both $P$ and $C$ to $f$ random peers (step ④). Upon the reception of $C$, peer $a$ validates the $C$ and persists the confirmation if it is valid. Both parties are now in possession of proposal $P$ and confirmation $C$ that together prove an agreement on work between these parties. The process of accounting work is lightweight since it requires minimal computational power and data exchange. Also, peers can engage in the recording of multiple interactions simultaneously.

A potential risk is that $b$ refuses to confirm $P$, even though the incoming proposal is valid and contains the correct work details. This could, for example, occur when confirming $P$ negatively impacts $b$'s social standing. This leaves $a$ with an unconfirmed proposal, which alone is not sufficient evidence to convince other peers of the performed work by $a$ for $b$. When $b$ refuses to sign an incoming proposal, $a$ will add $b$ to the local blacklist managed by applications, refusing to perform work for $b$ until $b$ has confirmed $P$. The losses for $a$ depend on the magnitude of the (unconfirmed) work performed for $b$. To minimize these losses, we suggest that decentralized applications record small units of works using ConTrib. For example, a file-sharing application can choose to account unconfirmed work when it reaches a threshold, e.g., 10 MB of traffic exchanged. Depending on the granularity of accounting, this approach can significantly reduce the impact of peers refusing to acknowledge the contributions of their counterparties.

### 3.4. Improving resilience by linking records

To prevent the modification of created records, we enforce each peer in ConTrib to link their records together in a *personal ledger*, incrementally ordered by creation time. Linking records will also make it harder for malicious peers to hide specific records. We make the following four modifications to records:

1. First, we include a sequence number $s \in \mathbb{Z}$ in each record that is incremented by one when a new record is added to one's personal ledger. The sequence number of the first record in the personal ledger is 1.
2. Second, each record now includes the hash of the prior record in the personal ledger of the creator. This modification makes the ConTrib data structure comparable to a hash chain, e.g., as used by blockchain applications. The modification of a particular record now changes the hash of subsequent records, a feature that enables us to detect illegitimate changes to stored records (also see Section 4). The previous hash of the first record in a personal ledger is empty and referred to as $\perp$.
3. Third, we extend the confirmation pointer with the sequence number of the proposal record that it confirms.
4. Fourth, we include at most $b$ additional hashes in each record of distinct, prior records in the same personal ledger. We indicate the set with these hashes with $S$ and call these hashes *back-pointers*. As we will further show in Section 4, the inclusion of these back-pointers significantly speeds up the detection of fraud. The required back-pointers in some record $R$ are deterministically given by a pseudo-random function $\sigma$ that takes the public key of the record creator and the sequence number of $R$ as input. $\sigma$ returns a set with at most $b$ prior records which hashes should be included in $R$. All peers must use the same version of $\sigma$, which we achieve by bundling its implementation in the ConTrib software.

The above modifications change the attributes of proposal and confirmation records. We re-define a proposal $P$, created by peer $a$ and with counterparty $b$, as follows:

$$P = (\texttt{pubKey}, \texttt{pubKeyOther}, \texttt{payload}, \texttt{sig}, \texttt{seqNum}, \texttt{prevHash}, \texttt{backPointers})$$

The variables coloured green are new compared to our previous definition of $P$. `seqNum` refers to the sequence number of $P$, `prevHash` indicates the hash of the previous record, and `backPointers` is the set with back-pointers (where $|S| \leq b$). We re-define a confirmation $C$, created by peer $b$, as follows:

$$C = (\texttt{pubKey}, \texttt{pubKeyOther}, \texttt{payload}, \texttt{linkInfo}, \texttt{sig}, \texttt{seqNum}, \texttt{prevHash}, \texttt{backPointers})$$

We extend confirmations with the same attributes as a proposal but replace the `proposalHash` attribute with `linkInfo`. This is in accordance with our third modification. `linkInfo` is now defined as a tuple with the hash and sequence number of the referred proposal record:

$$C.\texttt{linkInfo} = (\texttt{hash}, \texttt{seqNum})$$

Creating records yields the graph structure, as shown in Fig. 4. Fig. 4 shows a part of the ConTrib graph with six records, created by three distinct peers ($a$, $b$ and $c$). Same-coloured records are part of a single personal ledger, and arrows represent hash pointers to other records. Proposals have a solid border whereas confirmations have a dashed border. Note how the record in $a$'s personal ledger with sequence number 55 is unconfirmed. For presentation clarity, we only show the pointer to the prior record in one's personal ledger and omit additional back-pointers from the figure.

ConTrib publicly accounts work in interlinked personal ledgers. Since all performed work is publicly stored and accessible, other users might acquire and analyse ConTrib records to reveal potentially sensitive information, e.g., the time at which a particular user is online or the interaction patterns between users. To reduce this threat, we outline two techniques that applications can use to enhance privacy. First, applications can account performed and consumed work in *batches*. For example, an application can record all outstanding contributions and consumptions every hour, therefore hiding granular work statistics. Second, an application can add some noise to the amount of work being accounted (*fuzzy accounting*). This technique effectively reduces linkability, e.g., when accounting traffic that is being relayed through multiple hops. We believe that the combined power of these two techniques provides sufficient privacy guarantees for most decentralized applications. A more advanced approach, used by the Monero cryptocurrency, leverages ring signatures and zero-knowledge proofs to hide the amounts of work performed [46,47]. This approach would, however, require fundamental changes to ConTrib and we leave this enhancement for further work.

## 4. Detecting fraud

We require that ConTrib detects illegitimate tampering of the records in a personal ledger. ConTrib is built around fraud *detection* instead of *prevention*. We argue this is a reasonable assumption for two reasons. First, decentralized applications often do not require the prevention of fraud [48]. We argue that fraud prevention is disproportional

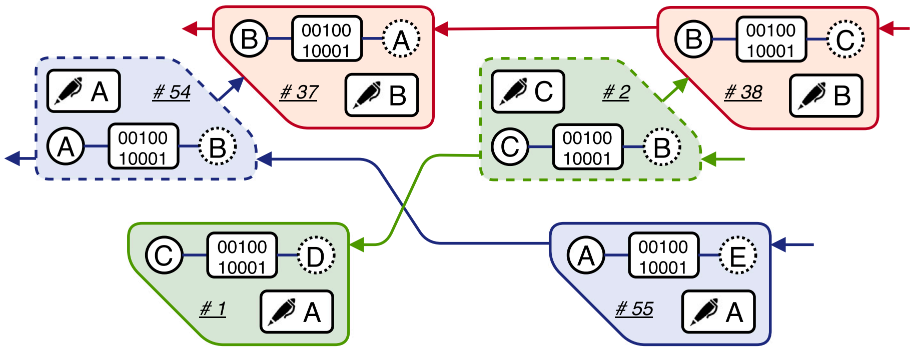


**Fig. 4.** A part of the ConTrib DAG, involving five peers and six records: four proposals (solid borders) and two confirmations (dashed borders). The proposal created by user $a$ with sequence number 55 is unconfirmed. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

in the context of work accounting. Second, preventing fraud is often a resource-intensive process that requires peers to reach a consensus on all created records, e.g., by using classical BFT algorithms or Proof-of-Work [49]. The requirement to reach consensus would severely limit the scalability of ConTrib.

Fraud in ConTrib occurs when a peer illegitimately modifies one of the records in their personal ledger. This fraud, for example, happens when an adversary attempts to hide a specific record in the personal ledger by replacing it with another one. This action would result in pairs of records with the same sequence number and the same creator, but with a different hash. This violates the integrity of the ConTrib data structure. A key objective of ConTrib is to detect such conflicting records quickly.

### *4.1. Detecting forks*

Fraud in ConTrib is detected by sharing newly created records with other peers, and by requesting random records in the personal ledgers of others. Each peer assesses the consistency of incoming records with the ones in its local database. This simple approach allows for quick detection of fraud through the collective effort of peers. In Fig. 5, we visualize four identified scenarios in which we can either expose an adversarial peer (scenario I and II) or detect an inconsistency without assigning blame (scenario III and IV). Each scenario shows the situation from a single peer's perspective and highlights records that a peer has in its local database, or does not have. Records not in the possession by a peer are faded. Records with the same colour are created by the same peer. We discuss each scenario and elaborate on how they either lead to fraud exposure or the detection of an inconsistency.

- **Scenario I**. The first scenario, visualized in Fig. 5a, describes a situation where a peer can directly expose a fork in the personal ledger of peer $a$. The personal ledger of peer $a$ has been forked since records $R_1$ and $R_2$ have the same sequence number but a different hash. As soon as another peer, say $b$, receives $R_1$ while already having $R_2$, or receives $R_2$ while already having $R_1$, the pair $(R_1, R_2)$ is sufficient evidence to expose the fraud by $a$. The digital signature by $a$ in the records prove that $a$ deliberately created both records. Note that $b$ does not need to have $R_3$ to detect nor to prove this fraud. We call the pair $(R_1, R_2)$ a *fraud proof*. Fraud proofs are by default shared with other peers in the network through a `FraudProof` message.
- **Scenario II**. The second scenario describes the situation where one can prove fraud by detecting inconsistencies in the included back-pointers of records. Fig. 5b shows four records created by peer $a$. Records $R_3$ and $R_4$ contain the hash of the record with sequence number 54 in the back-pointer set; however, these back-pointers describe the same record with a different hash. The pair $(R_3, R_4)$ is irrefutable proof that peer $a$ has committed fraud and can be used to construct a fraud proof.

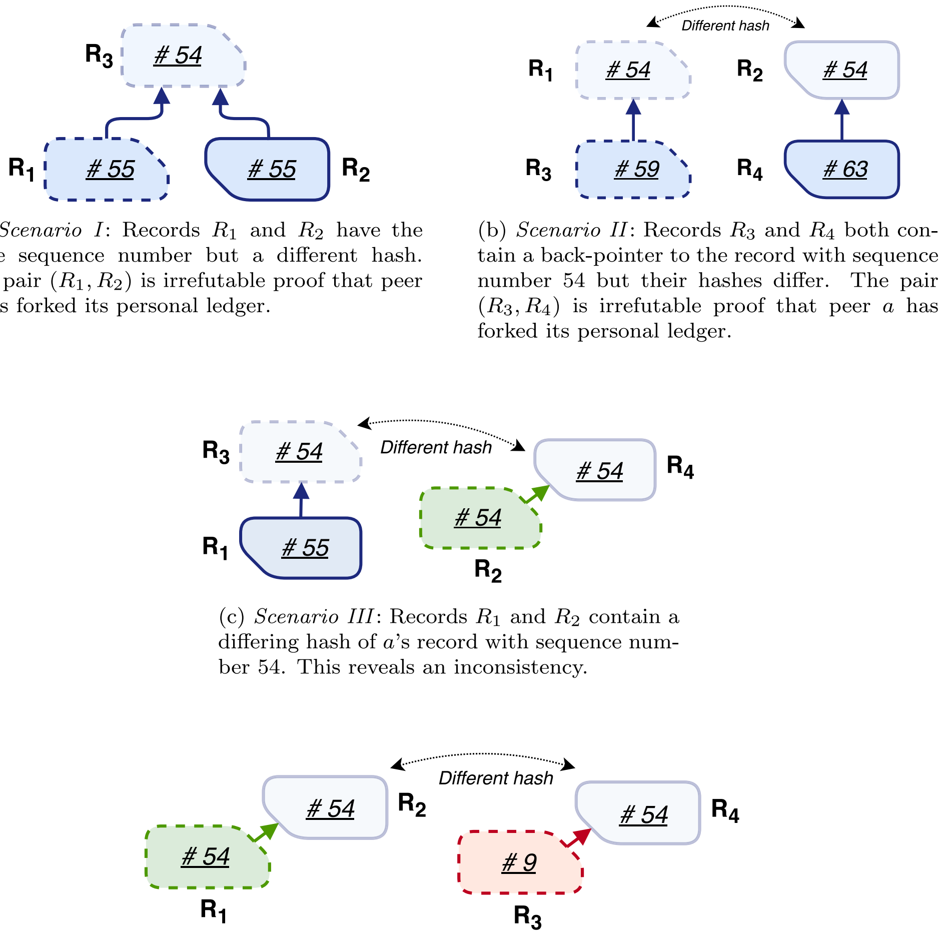
**Algorithm 1** The validation of the fields in record $R$.

```
procedure VALIDATEFIELDS($R$)                                ▷ Step 1
    $valid \leftarrow$ true
    if $R.seqNum < 1$ then
        $valid \leftarrow$ false
    if ISCONFIRMATION($R$) and $R.linkInfo.seqNum < 1$ then
        $valid \leftarrow$ false
    if not PUBLICKEYISVALID($R.pubKey$) then
        $valid \leftarrow$ false
    if not SIGNATUREISVALID($R.pubKey$, $R.sig$) then
        $valid \leftarrow$ false
    if not PUBLICKEYISVALID($R.pubKeyOther$) then
        $valid \leftarrow$ false
    if $R.seqNum = 1$ and $R.prevHash \neq \perp$ then
        $valid \leftarrow$ false
    return valid
```

- **Scenario III**. Fig. 5c shows a third scenario where a peer receives proposal $R_1$ and already has confirmation $R_2$, or receives confirmation $R_2$ while already having proposal $R_1$. The peer does not have records $R_3$ and $R_4$. The hash of record $R_3$ in $R_1$ differs from the hash in the confirmation pointer in $R_2$. This situation reveals an inconsistency that is either introduced by peer $a$ forking its personal ledger at height 54, or by $b$ having included a wrong hash in $R_2$. To assign blame, the peer that is validating the incoming record requires either $R_3$ or $R_4$. A peer that encounters this situation sends the pair $(R_1, R_2)$ within a `Inconsistency` message to other random peers, hoping that others will be able to expose the malicious peer.
- **Scenario IV**. Fig. 5d highlights the fourth scenario where a peer either receives confirmation $R_1$ while already having confirmation $R_3$, or vice versa. Both confirmations point to a record with the same public key and sequence number, but the hash of this record differs. This situation either indicates a fork of the personal ledger of $a$, or it can be the result of an invalid pointer in one of the confirmations. Similar to scenario III, the validating peer sends the pair $(R_1, R_3)$ within a `Inconsistency` message to other, random peers.

### *4.2. Record validation logic*

Based on the four identified scenarios, we design and describe the validation logic of an incoming record $R$. Each peer keeps track of known hashes in a dictionary named `knownHashes`. This dictionary is indexed with a tuple, containing the public key and sequence number of a record. The value of dictionary entries is the hash of the record



(a) *Scenario I*: Records $R_1$ and $R_2$ have the same sequence number but a different hash. The pair $(R_1, R_2)$ is irrefutable proof that peer $a$ has forked its personal ledger.

(b) *Scenario II*: Records $R_3$ and $R_4$ both contain a back-pointer to the record with sequence number 54 but their hashes differ. The pair $(R_3, R_4)$ is irrefutable proof that peer $a$ has forked its personal ledger.

(c) *Scenario III*: Records $R_1$ and $R_2$ contain a differing hash of $a$'s record with sequence number 54. This reveals an inconsistency.

(d) *Scenario IV*: Records $R_1$ and $R_3$ both confirm $a$'s record with sequence number 54, but they contain a different hash. This reveals an inconsistency.

**Fig. 5.** Four scenarios that allows a peer to either expose fraud (forking of a personal ledger), or to detect an inconsistency (without assigning blame). The colour of each record indicates the identity of its creator (blue for $a$, green for $b$ and red for $c$). Solid and dashed records indicate proposals, respectively confirmations. Opaque records are not in possession by the peer. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

**Algorithm 2** The consistency validation of hashes in an incoming record against known ones.

```
 1: procedure VALIDATEHASHES(R)                                   ▷ Step 4
 2:     $hash \leftarrow knownHashes[(R.pubKey, R.seqNum - 1)]$
 3:     if $hash \neq \perp$ and $hash \neq R.prevHash$ then
 4:         return false
 5:
 6:     for $seqNum, hash$ in $R.backPointers$ do
 7:         $known \leftarrow knownHashes[(R.pubKey, seqNum)]$
 8:         if $known \neq \perp$ and $hash \neq known$ then
 9:             return false
10:     return true
```

being queried. The validation logic of incoming records consists of the following five steps:

**Step 1.** We first verify the validity of the fields in incoming record $R$. This step is performed by the VALIDATEFIELDS procedure, which returns a boolean value indicating whether the fields in the record are valid or not. A pseudocode description of this procedure is given in Algorithm 1. This step validates the sequence number (line 3), the included public keys (line 7 and 11), and the digital signature (line 9). If the incoming record is a confirmation, it also verifies that the sequence number in the `linkInfo` attribute is within a valid range (line 5). It also checks whether the hash of the prior record is sane when the record is the first in ones personal ledger (line 13). This step does not compare the validity of $R$ within the context of other records. Any error in the included fields of $R$ is computationally efficient to detect and likely originates from a software bug.

**Step 2.** Next, we query the database for a record with the same public key and sequence number as the incoming record $R$. If such a record $R'$ is in the database, we check the equality of $R$ and $R'$ by performing a comparison between their included fields. If $R \neq R'$, we have detected a fork in the personal ledger of the creator behind $R$. We then share the fraud proof $(R, R')$ with other peers in the network. During this step, we detect the fraud described by scenario I in Section 4.1.

**Step 3.** Then, we verify if the hash pointers in the incoming record are consistent with known ones. This is performed by the VALIDATEHASHES procedure which pseudocode description is given in Algorithm 2. This procedure first checks whether the `prevHash` attribute in $R$ is consistent with the information in the `knownHashes` dictionary (line 3). We then iterate over all included back-pointers and verify the consistency of these hashes with the entries in the `knownHashes` dictionary (line 6–10). We can detect the inconsistency described by scenario II during this step.

**Step 4.** Next, we compare incoming record $R$ with a link record, if such a record is available in the database. When $R$ is a proposal,

**Algorithm 3** The validation of an incoming record against a linked record.

```
1:  procedure validateLink(R)                                         ▷ Step 3
2:      $linked \leftarrow db.$getLinked(R)
3:      if $linked = \bot$ then
4:          return true
5:
6:      $proposal \leftarrow linked$ if isConfirmation(R) else $R$
7:      $confirmation \leftarrow linked$ if isConfirmation($linked$) else $R$
8:
9:      if $confirmation.pubKeyOther \neq proposal.pubKey$ then
10:         return false
11:     if $confirmation.linkInfo.seqNum \neq proposal.seqNum$ then
12:         return false
13:     if $confirmation.linkInfo.hash \neq proposal.hash$ then
14:         return false
15:     $linkLinked \leftarrow$ db.getLinked($linked$)
16:     if $linkLinked \neq \bot$ or $link\_linked \neq R$ then
17:         return false
18:     return true
```

we get the corresponding confirmation from the database, and if $R$ is a confirmation, we get the corresponding proposal. This step is performed by the validateLink procedure which pseudocode description is given in Algorithm 3. We first get the linked record from the database (line 2) and only continue with this validation step if we have this record in the database. We check whether the public keys included in the proposal and confirmation are consistent (line 9), and verify the consistency of the `linkInfo` attributes in the confirmation (line 11–14). We detect the inconsistency described by scenario III during this step (line 15–17).

**Step 5.** Finally, we verify the validity of the included payload, which is an application-dependent validation procedure. As we will further outline in Section 5, decentralized applications using ConTrib should implement a *validation* policy that denotes whether the payload of an incoming record is valid in the context of the connected application.

If any of the above steps fail, the record is considered invalid.

### 4.3. Exchanging records with other peers

The detection of fraud in ConTrib depends on peers exchanging records with each other. A peer is motivated to share collected records with others since they might eventually reveal fraud conducted by one of their former counterparties. So far, we have not discussed how records are disseminated. Record dissemination is an essential process that affects the speed at which fraud can be detected. For example, a slow record exchange strategy is likely to increase fraud detection times compared to more aggressive record dissemination. We consider both *push-based* and *pull-based* exchange of records, which is explained next.

**Pull-based record exchange.** Each peer by default requests (pulls) records from other random peers at a fixed rate by sending out `Request` messages. Applications can choose to send out `Request` messages to specific peers to build profile information about that peer, e.g., to detect free-riders. A `Request` message contains a list of sequence numbers that the recipient should send back. When receiving a `Request` message, the recipient also includes linked proposal or confirmation records in the response. When a peer $a$ does not respond with records within a reasonable time, the requesting peer adds $a$ to the local blacklist.

**Push-based record exchange.** ConTrib also supports push-based record exchange, where the creator of a record disseminates it to $f$ random other peers (as also discussed in Section 3.3). This push-based exchange allows for quick detection of fraud since the probability of no user receiving two conflicting records goes to zero quickly, even when the network size increases [50]. Even if the malicious peer refrains from broadcasting a conflicting record, the counterparty is very likely to do so, assuming there is no collusion between interacting peers. Immediate dissemination of created records in the network also increases record availability when the sending peer goes offline.

### 4.4. Limitations

Even though our simple algorithm can detect the modification of records, the probabilistic nature of our algorithm can render ConTrib unsuitable for deployment in specific application domains. Since fraud detection is a probabilistic approach, some fraud instances can take relatively long to be uncovered (e.g., several minutes). We also observed this during our experiments (see Section 6). At the same time, we argue that this is not an insurmountable problem in decentralized applications where performed work holds no or low monetary value.

Since our algorithm is based on fraud *detection*, ConTrib is not suitable for applications that require a high level of security, as is the case with decentralized financial applications. We believe that blockchain technology provides more appropriate security guarantees for such application domains, at the cost of increased resource usage and lower scalability.

Finally, we note that ConTrib is mainly built for the lightweight accounting of work in decentralized applications. In its current state, ConTrib cannot capture more complicated operations, e.g., executing arbitrary logic like smart contracts. However, as demonstrated by recent research advancements, a lightweight accounting mechanism can be an enabling component to devise novel types of decentralized applications with dynamic risk guarantees [51–53].

## 5. System architecture

We devise a system architecture of our ConTrib mechanism, see Fig. 6. The network layer is the lowest layer in our architecture and provides the primitives for decentralized communication and peer discovery. This layer can be realized using existing frameworks to build peer-to-peer overlay networks, for example, libp2p.[2]

**Record manager.** The record manager interacts with the network layer to disseminate records and processes incoming ones. It queues incoming records for validation and persists incoming fraud proofs to the database and connected application. It also manages the confirmation of incoming and valid proposals targeted at that peer. Applications using ConTrib should provide a *confirmation policy* that predicates whether an incoming proposal should be confirmed.

**Validator.** The validator determines the validity of incoming records according to the algorithms described in Section 4.2. Connection applications can provide a custom *validation policy*. If provided, this validation policy is invoked during step 5, when the application-specific payload in a record is validated. The flexibility to provide custom validation and confirmation policies for incoming records makes ConTrib universal and reusable across different application domains.

**Persistence.** Records and fraud proofs are persisted in a database. The ConTrib system architecture provides an interface for the queries made to the database and supports different database architectures, e.g., structured or unstructured models. Our system architecture includes a record *cache*, which is an intermediary component that stores all records in the personal ledger of the operating peer in memory. This cache allows ConTrib to quickly respond to incoming record queries in the personal ledger of the operating peer. This cache forwards queries to the database for the retrieval and storage of records and fraud proofs.

To contain the growth of the database and to keep the storage overhead manageable, an application can choose to periodically prune the ConTrib database when a storage threshold is reached. In our

[2] See https://libp2p.io.

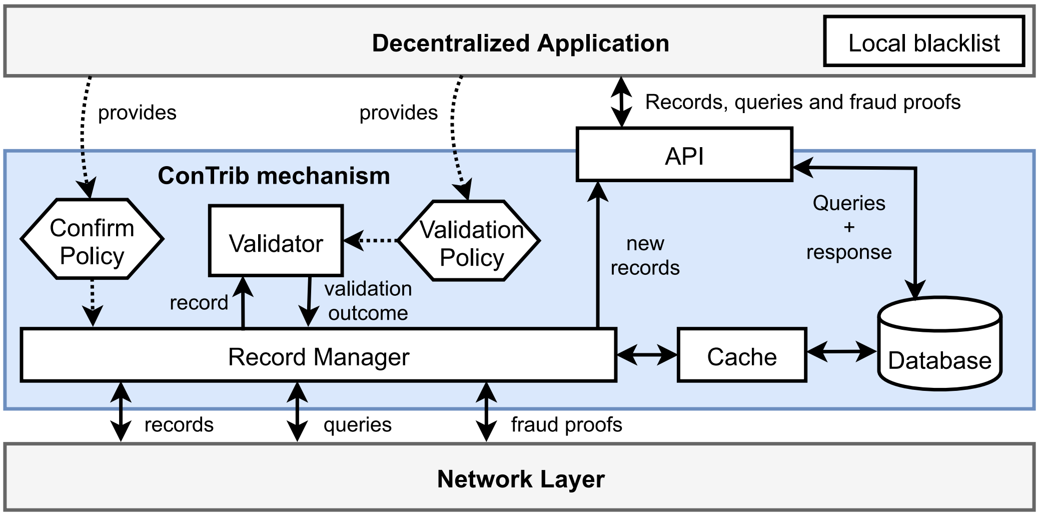


**Fig. 6.** The system architecture of ConTrib.

**Table 1**
The default parameters used during our evaluation.

| Parameter | Default value |
|---|---|
| Peers ($n$) | 1000 |
| Workload | 1 proposal per second per peer |
| Record exchange strategy | `PULL+RAND+PUSH` |
| Record fanout ($f$) | 5 |
| Record request batch size | 2 |
| Record request interval | 0.5 s |
| Packet loss rate | 0% |
| Individual forking probability | 10% |
| Back-pointers ($b$) | 10 |

implementation, by default, we start pruning when at least one million records have been stored. Applications may increase or decrease this number, depending on the storage capacities of participating peers and the deployment environment. The default pruning strategy of ConTrib continuously removes the record with the lowest database insertion timestamps until the database size has reached its storage threshold again. The pruning of older records might cause some forks to go undetected since records are removed before a fraud proof can be constructed. As we will show in Section 6.2, most forks in ConTrib are quickly detected, and there should be ample time to detect inconsistencies before relevant records are pruned.

**Fraud management.** When the validation algorithm exposes fraud, or when ConTrib receives an incoming fraud proof, the connected applications are notified of the fraud and can punish the misbehaving peer accordingly. For example, a fraud policy in a bandwidth sharing application could decide to not serve the fraudster for some time. The decentralized application store the digital identities of fraudsters in a local blacklist.

**Interactions between ConTrib and applications.** Decentralized applications interact with ConTrib through an API. This API allows connected applications to query the content of the database. Furthermore, connected applications can subscribe to incoming records. The record manager forwards new records to the API, which passes these records to subscribed applications.

## 6. Implementation and evaluation

Within this section we systematically explore how ConTrib behaves when modifying system parameters. We implement ConTrib in the Python 3 programming language. We leverage the network library implemented by our group, and use the UDP protocol for network communication.[3] Our implementation uses the `asyncio` framework for asynchronous event handling. The implementation features both an in-memory storage for experimentation and a persistent (sqlite) database which can be used to persist records over different sessions. The full implementation of ConTrib, including unit tests and documentation, is published on GitHub.[4]

### 6.1. *Experiment setup*

We evaluate the impact of different parameters on the efficiency of fraud detection. We do so by measuring the time between committing the fraud and its initial detection. We substitute our networking layer with the SimPy discrete event simulator [54]. Each peer in the ConTrib network knows the network address of 100 random other peers, resulting in an unstructured overlay topology. Table 1 lists the default parameters during our experiments. To encourage reproducibility, we have open-sourced the ConTrib simulator and all experiment scripts.[5]

**Workload and attack model.** During our experiments, peers create records with other random peers. Our default workload has each peer initiate one proposal per second with another random peer. Note that the rate at which new records are created grows with the network size, which should capture the dynamics of real-world applications (when there are more peers, there is usually more work performed in the application). We use a uniform transaction load to analyse the characteristics of ConTrib under a predictable load. Even though this transaction load is predictable, it resembles an application where work is periodically accounted. We experiment with network sizes ranging from 1000 to 10’000 online peers. Though some deployed networks have many more peers (e.g., BitTorrent and Tor), our experimental results suggest that ConTrib has no issues scaling beyond 10’000 peers. In Section 6.6, we subject ConTrib to a realistic workload, extracted from the interactions in a decentralized file-sharing application.

Each peer forks its personal ledger with a probability of 10% by removing the last record in its personal ledger and re-using its sequence number to create a new record. Each peer commits this fraud once. A peer committing fraud will not broadcast the duplicate record when push-based record exchange is enabled. In each experiment run, all peers start with an empty personal ledger, and interaction partners

[3] See https://github.com/tribler/py-ipv8.

[4] See https://github.com/tribler/py-ipv8/tree/master/ipv8/attestation/trustchain.

[5] See https://github.com/tribler/trustchain-simulator-pysim.

always confirm incoming proposals. A peer that has exposed the fraud of another peer will refuse to confirm the proposals by that peer. Each experiment run terminates either when all fraud attempts have been discovered or after ten minutes have elapsed. We are interested in the detection time of fraud instances, which is the time period between committing the fraud and its first detection by a peer in the network.

**Record exchange strategies.** We consider the following four strategies for exchanging records. With the PULL strategy, each peer requests two contiguous records at a random height in the personal ledger of another random peer every half a second (the *record request batch size* and *record request interval* parameters in Table 1). Under the PULL+RAND strategy, a peer also returns five random records in their database upon a query. Including random records in responses enables the detection of fraud of offline peers. The PULL+PUSH strategy also pushes new records to $f$ random users upon creation, compared to the PULL strategy. Finally, we consider the PULL+RAND+PUSH record exchange strategy, which is a combination of the above techniques.

### 6.2. Scalability

Our first experiment quantifies the scalability of ConTrib when increasing the number of peers in the network, see Fig. 7. Fig. 7a shows the effect of increasing the number of peers on the average time until fraud detection, for the four discussed record exchange strategies. For all record exchange strategies, the average fraud detection time seems to increase when the network size grows. For $n = 10'000$, the PULL strategy shows an average fraud detection time of 63.5 s, whereas this number decreases to 3.6 s under the PULL+RAND+PUSH strategy. We notice that the PULL+PUSH and PULL+RAND+PUSH strategies show detection times under five seconds on average. These low detection times demonstrates that disseminating a record just after its creation is a highly successful strategy. Including random records in crawl responses also seems to decrease the average fraud detection times. For $n = 10'000$, the average fraud detection time decreases from 63.5 s for the PULL strategy to 27.8 s for the PULL+PUSH strategy.

In Fig. 7b, we show the Empirical Cumulative Distribution Function (ECDF) of fraud detection times for $n = 5000$. We observe that it can take several minutes for some fraud attempts to be discovered, in particular for the PULL and PULL+RAND strategies. This is not unexpected since fraud detection in ConTrib is a highly probabilistic approach. For the PULL strategy, we can detect 90% of fraud attempts within 160 s and 50% of the fraud attempts within 30 s. We also observe that the vast majority of fraud is detected within a few seconds when pushing random records after creation. 82.9% of all fraud attempts are detected within five seconds for the PULL+RAND+PUSH strategy.

Fig. 7c shows the average network usage per-peer as the network size increases, for different record exchange strategies. The PULL strategy requires less than 10 KB per second of network usage. Compared to the PULL strategy, the PULL+RAND+PUSH strategy requires more than three times as much bandwidth, around 35 KB per second. Note how the network usage remains roughly the same for all considered record exchange strategies as we add more peers to the experiment. Fig. 7c also shows that it is feasible to deploy ConTrib in consumer-grade network environments. System designers can decrease the network usage of ConTrib even more by lowering the crawl interval or crawl batch size, at the cost of increased fraud detection times.

Not all fraud has been detected after our experiment has ended. Fig. 7d shows the percentage of fraud attempts that has been detected after the experiment has ended, for an increasing number of peers and different record exchange strategies. It becomes less likely that fraud is detected during our experiment when increasing the network size under the PULL strategy. For $n = 10'000$, 7.28% of fraud attempts remain undetected. These attempts would likely be discovered when prolonging the experiment duration.

### 6.3. Packet loss

We reveal the robustness of ConTrib by quantifying the effect of packet loss on the efficiency of fraud detection. To this end, we increase the packet loss rate, up to 20%, and run our simulations under our four record exchange strategies. Even though a packet loss rate of 20% is unlikely for any environment in which ConTrib is to be deployed, we want to estimate how robust ConTrib is, even in such extreme circumstances. We expect fraud detection times to increase when network stability is lower since losing packets makes it less likely to detect inconsistencies.

Fig. 8 shows the average fraud detection times when increasing the packet loss rate for our four record exchange strategies. We observe that fraud detection times increase under the PULL and PULL+RAND strategies, whereas this effect is less for the PULL+PUSH and PULL+RAND+PUSH strategies. Average fraud detection times, under the PULL strategy, increase from 44.6 s with no packet loss to 94.1 s with 20% packet loss. For the PULL+RAND+PUSH strategy, this same increase is from 2.3 s to 6.0 s. We notice that with a packet loss of 20% and the PULL strategy, 23.8% of all fraud attempts remains uncovered after the experiment has ended. This number is just 0.2% when no packets are lost. Under the PULL+RAND+PUSH strategy, we see that all fraud attempts are discovered in our experiment, for all evaluated packet loss rates.

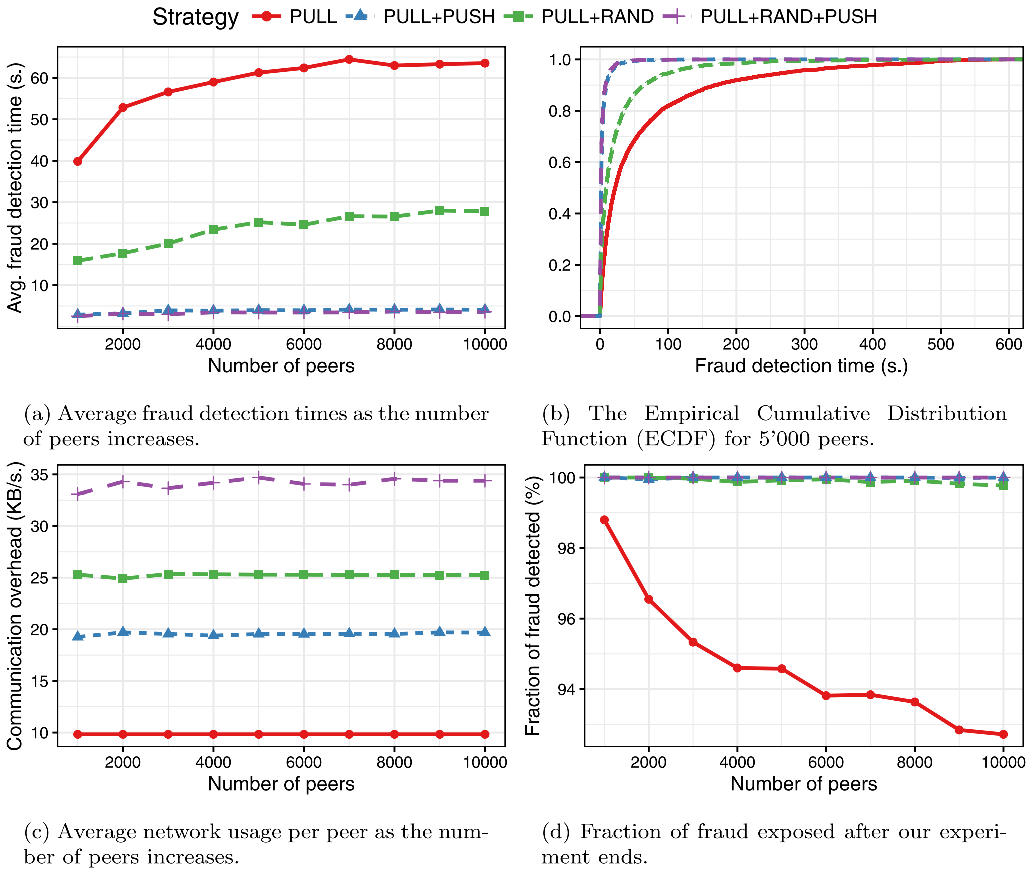
### 6.4. Request interval and batch size

We modify the record request interval and batch size and analyse the effect on the average fraud detection times, see Fig. 9. Fig. 9a visualizes the impact of the record request interval on the average fraud detection times for different record exchange strategies. We increase the record request interval, ranging from 0.5 s to 5 s, in steps of 0.5 s. First, we observe that the average fraud detection times for the PULL+PUSH and PULL+RAND+PUSH strategies remains roughly constant. This trend indicates that pushing records to random peers is the dominant logic of the record exchange strategy, very effectively exposing fraud instances. We also note that average fraud detection times for the PULL and PULL+RAND strategies are increasing when the record request interval becomes larger. For the PULL strategy, we notice that when the request interval is over 3 s, most fraud instances remain undetected after our experiment finishes. These fraud instances become increasingly harder to detect as the modified record in a personal ledger gets “buried” under additional records. As such, the average fraud detection time is likely higher when running the experiment until all fraud instances have been detected. Unfortunately, we are unable to significantly prolong the experiment duration as our simulations are already using peak memory usage, despite various optimization efforts.

Fig. 9b shows the average fraud detection times when increasing the number of records queried in each request from 1 to 10, for different record exchange strategies. Again, the PULL+PUSH and PULL+RAND+PUSH strategies are indifferent to this increase. The average fraud detection time for the PULL strategy decreases from 60.4 s to 12.4 s when increasing the request batch size from 1 record to 10 records, respectively. This decrease indicates that increasing the request batch size is particularly effective when using the PULL strategy, at the expanse of increased bandwidth usage.

### 6.5. Back-pointers

We vary the number of back-pointers ($b$) in each record and analyse the effect on average fraud detection times. We suspect that adding more back-pointers increases the probability of detecting fraud since individual records now bear more hashes of records in ones personal ledger. This advantage comes, however, at a cost of additional network usage and computational overhead to analyse the back-pointers. Each back-pointer adds 32 bytes to the serialized record size.



(a) Average fraud detection times as the number of peers increases.

(b) The Empirical Cumulative Distribution Function (ECDF) for 5’000 peers.

(c) Average network usage per peer as the number of peers increases.

(d) Fraction of fraud exposed after our experiment ends.

**Fig. 7.** The results of our scalability experiments, with up to 10’000 peers. We evaluate four record exchange strategies.

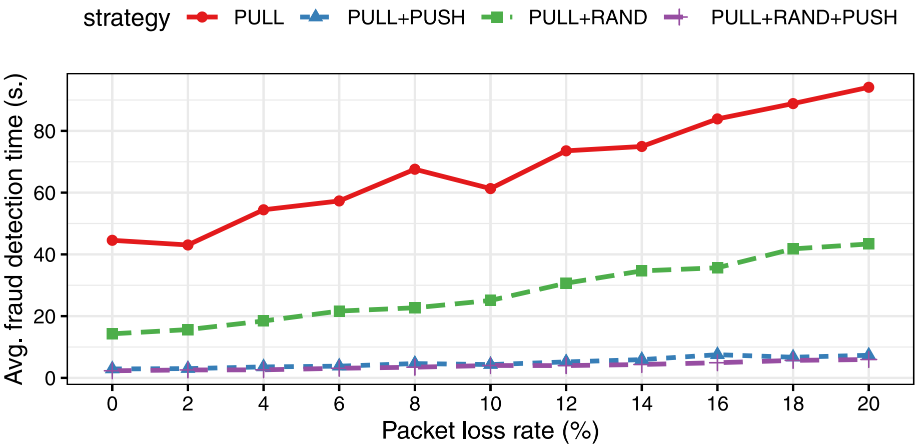
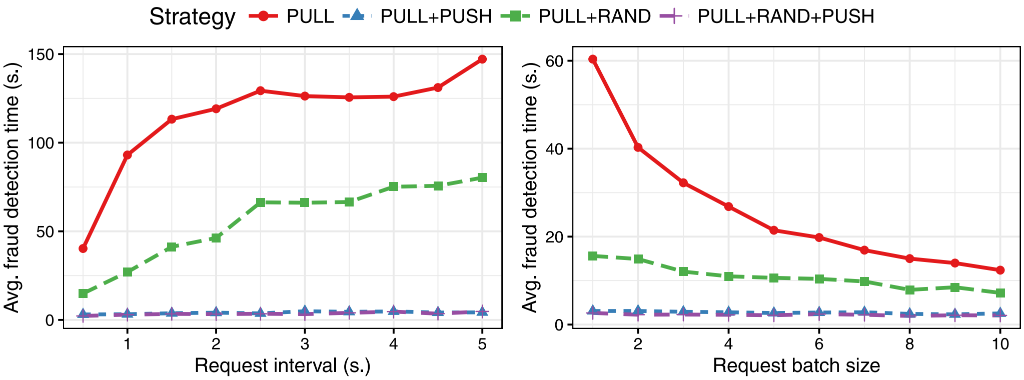


**Fig. 8.** The effect of packet loss on the average fraud detection times, for different record exchange strategies.

Fig. 10 shows the average time until fraud is detected while varying the number of back-pointers and considering different record exchange strategies. Adding additional back-pointers indeed decreases fraud detection times. Under the PULL record exchange strategy, it takes 111.2 s to detect fraud when no additional back-pointers are included. In comparison, this number decreases to 41.6 s when adding up to ten back-pointers to each record (a decrease of 58.4%). This decrease is much more for the PULL+PUSH strategy, namely 97%. Note that the effect of adding more back-pointers diminishes for $b > 4$. This effect can likely be attributed to the fact that all peers start with an empty personal ledger in our simulations, and that different records with lower sequence numbers in the same personal ledger are more likely to include identical hashes in their back-pointers. However, we believe that the effect of additional back-pointers becomes more apparent when personal ledgers grow to considerable sizes, since different records are then more likely to include more unique hashes.

### 6.6. *Fraud detection times under a realistic workload*

Our experiments conducted so far are using a synthetic workload. We now evaluate the effectiveness of fraud detection in ConTrib of our four considered record exchange strategies using a realistic workload. This workload is derived from deployment data of ConTrib in Tribler, our decentralized file-sharing application. An interaction describes



(a) Average fraud detection times as the request interval increases.

(b) Average fraud detection times as the request batch size increases.

**Fig. 9.** The effect of changing the request interval batch size on the average fraud detection times.

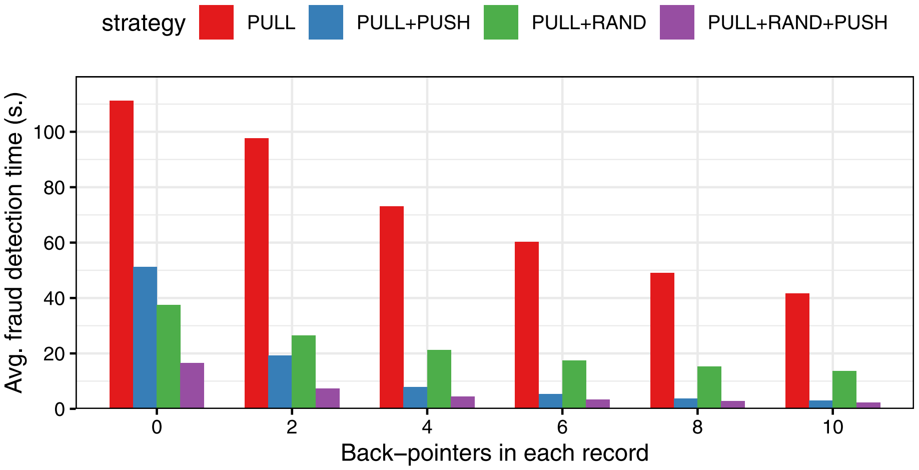


**Fig. 10.** The effect of adding more back-pointers to each record on the average fraud detection times, for different record exchange strategies.

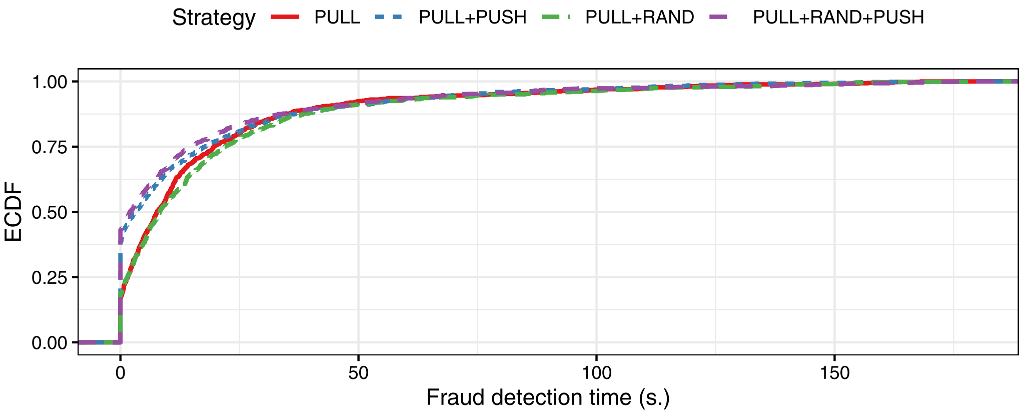


**Fig. 11.** Fraud detection times lower than 180 s for different record exchange strategies when replaying a day of interactions made by Tribler users.

network communication between two peers in a Tor-like overlay. We provide further details on this dataset in Section 7. The following experiment replays interactions made during the busiest 24 h of our deployment period: November 28, 2020. On this particular day, a total of 440’130 records have been created, involving 2027 digital identities. In the following experiment, we simulate a peer for each digital identity. In line with our prior experiments, each peer commits fraud with a probability of 10% when creating a new record. To match the ConTrib settings in our deployed environment (see Section 7), we increase the record request interval to 10 s.

We noticed that all fraud instances have been detected after our experiment ends. The average fraud detection time for the PULL strategy is just 29.4 s whereas this number decreases to 18.5 s for the PULL+RAND+PUSH strategy. 2.5% of all fraud instances take longer than three minutes to detect, with the highest detection time being 1276 s (just over 21 min). Fig. 11 shows the Empirical Cumulative Distribution Function (ECDF) of fraud detection times, for the evaluated strategies. For presentation clarity, we only show the detection times of fraud instances lower than three minutes. We observe that over 25% of all fraud for the PULL+PUSH strategy is detected immediately. During this experiment, we also see that the network usage per peer is relatively low. For the PULL strategy, each peer, on average, consumes merely 1.7 MB of hourly network traffic. This number increases to 5.1 MB under the PULL+RAND+PUSH strategy.

This experiment shows that ConTrib, under a realistic workload of Tribler interactions, exhibit relatively low fraud detection times and has low network usage. As we have also measured during our deployment trial (see Section 7), the resource overhead of ConTrib is minimal. For Tribler, the fraud detection times shown in Fig. 11 are acceptable. However, as we have also shown in our other experiments, these detection times can further be improved with more frequent record crawling.

### 6.7. Discussion

In summary, our experiments demonstrate that ConTrib exhibits low fraud detection times, scales when the network grows, has reasonable bandwidth overhead, and is robust against packet loss. We have also demonstrated the effect of the number of back-pointers in each record on the efficiency of fraud detection. Finally, we have shown that ConTrib remains effective at detecting fraud when using a realistic workload. Even though we have not evaluated the effect of all parameters in Table 1, we believe that this set of experiments provides a good starting point for system designers to adopt and configure ConTrib. With our open-source simulator, system designers can quickly analyse the effect of different parameters, guided by a synthetic or realistic workload that resembles the behaviour in their application.

We have demonstrated that there is a trade-off between the average fraud detection times and network usage. The acceptable network usage differs per applications. For example, bandwidth is less likely to be a bottleneck when considering a video streaming application compared to an Internet-of-Things environment with low-resource devices. By reducing the record request intervals, fanout value, and the maximum number of back-pointers, one can reduce the bandwidth footprint of ConTrib, at the cost of increased fraud detection times. Fig. 7c shows that the active record exchange strategy has a notable impact on network usage. In a dynamic network where the sessions of peers are short-lived, we recommend the PULL+RAND or PULL+RAND+PUSH strategies, where peers share random records in their database with others. This strategy allows the detection of fraud attempts of offline peers. We recommend the PULL+RAND+PUSH strategy when fraud must be detected quickly. We recommend the PULL strategy when bandwidth is a limiting factor.

## 7. Applying ConTrib to address free-riding at scale

To show the effectiveness of ConTrib with a real-world application, we conduct a large-scale deployment trial of ConTrib with Tribler. Tribler is our decentralized file-sharing application and is downloaded by over 1.8 million users [14]. This application features an onion-routing overlay that tunnels BitTorrent traffic through *relay and exit nodes* to provide anonymity. Unfortunately, the Tribler network suffers from an undersupply of exit nodes, leading to frequent network congestions and overall degradation of download speeds for all users. We employ ConTrib to account the performed work by relay and exit nodes, and the consumed work by downloaders. We then punish free-riding behaviour by offering users with higher net contributions preferential treatment during periods of congestion. In the remainder of this section, we elaborate on the integration of ConTrib in Tribler, on the collected data, and on the effectiveness of free-riding detection.

### 7.1. Accounting bandwidth transfers

We enable peers to earn *bandwidth tokens* by providing services as a relay or exit node in the Tribler network. The mutations in bandwidth token balances of each peer is accounted in ConTrib records. For example, a payout corresponding to a data exchange of a 50 MB file between peers $a$ and $b$ deduces 50 MB of $a$'s balance and increments $b$'s balance by 50 MB (MB is the unit of this bandwidth token). Peers can have a negative balance of bandwidth tokens, in which case they have received more traffic than they contributed to the network.

When a peer downloads content using Tribler, the Tribler software establishes a *circuit*, containing exactly one exit node and optionally some relay nodes. This is comparable to circuits in the Tor protocol. All traffic is securely routed through relay and exit nodes. Fig. 12 shows how bandwidth tokens are paid out after a peer has downloaded a 50 MB file over a three-hop circuit (with two relay nodes and one exit node). The downloader accounts a transfer of 250 MB to the first relay using ConTrib. Specifically, the downloader creates a proposal record, containing the pair-wise byte counter with the first relay and the magnitude of the current payout. In the latest version of Tribler, newly created records are by default disseminated to 20 random peers. Each peer also requests a random record from another known peer every 10 s. During our deployment period, we have periodically revised the record exchange strategy in response to the network behaviour and growth.

After the downloader has finished the payout to the first relay, the first relay then transfers 150 MB to the next relay, resulting in a net positive token balance of 100 MB for the first relay. The rationale behind our payout scheme is that we reward relay and exit nodes for performing the cryptographic operations on the forwarded data. Relays that do not forward the payout to the next hop will be added to the local blacklist by the previous hop, therefore lowering their opportunity to earn bandwidth tokens.

Since this use-case involves anonymous downloading, there is an important trade-off between accountability and anonymity. We plan to address privacy concerns around the accounting of bandwidth transfers by having each peer aggregate and delay payouts. This privacy-enhancing technique is introduced in the work of Palmieri et al. [55]. Still, work accounting with ConTrib does not leak the identity of a downloader to other peers in the network, nor reveals any data being transferred over circuits. To address the uncontrolled minting of bandwidth tokens by accounting fake work, we are currently looking into the design and deployment of a Sybil-resistant reputation mechanism [36].

### 7.2. Circuit assignment

We use the bandwidth token balances included in ConTrib records to grant preferential treatment to dedicated peers during periods of congestion. Specifically, we modify the Tribler protocol such that each relay and exit nodes maintain a fixed number of *slots*. A circuit that includes a relay or exit nodes consumes one slot at their side. We distinguish between *random* and *competitive* slots. Random slots are filled on a first-come-first-serve basis whereas the assignment of competitive slot is based on the bandwidth token balance of a circuit initiator. The intuition behind this approach is to still give peers with lower trust scores an opportunity to earn a (random) slot but at the same time, give preferential treatment to well-behaving peers with competitive slots. The total number of such slots can be changed, depending on the hardware capabilities of the node operator.

A pseudocode description of the slot assignment logic is given in Algorithm 4. When a circuit initiation request arrives, the onCircuitRequest method is invoked and Tribler first determines if there is a random slot available (line 5–9). If so, we assign the new circuit to the random slot (line 7). If no random slot is available, Tribler queries the bandwidth token balance of the circuit initiator $i$ by requesting

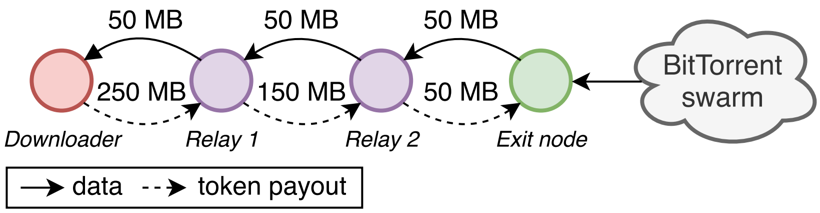


**Fig. 12.** Accounting specifications of an anonymous 50 MB BitTorrent download over a three-hop onion-routing circuit.

**Algorithm 4** The assignment logic of slots to circuits. *numRand* and *numComp* represent the maximum number of random and competitive slots, respectively.

```
 1: randomSlots ← [⊥] ∗ numRand
 2: competitiveSlots ← [(−∞, ⊥)] ∗ numComp
 3:
 4: function onCircuitRequest(circuit)
 5:     for $i = 0$ to numRand do
 6:         if randomSlots[$i$] = ⊥ then
 7:             randomSlots[$i$] ← circuit
 8:             return
 9:     Query the balance of the initiator of the circuit
10:
11: function onBalance(circuit, balance)
12:     lowestBalance ← ∞
13:     lowestIndex ← ∞
14:     for $i = 0$ to numComp do
15:         if competitiveSlots[$i$] = (−∞, ⊥) then
16:             competitiveSlots[$i$] ← (circuit, balance)
17:             return
18:         if competitiveSlots[$i$][0] < lowestBalance then
19:             lowestBalance ← competitiveSlots[$i$][0]
20:             lowestIndex ← $i$
21:     if balance > lowestBalance then
22:         destroyCircuit(competitiveSlots[lowestIndex][1])
23:         competitiveSlots[lowestIndex] ← (circuit, balance)
```

the records in the personal ledger of $i$. When receiving these records, Tribler determines the current bandwidth token balance and checks eligibility for a competitive slot (line 12–23). If there is an unoccupied competitive slot, Tribler assigns the new circuit to it (line 16). If all competitive slots are filled, the circuit of the initiator with the lowest amount of bandwidth tokens, say $p$, is destroyed if the token balance of $i$ is higher than the token balance of $p$ (line 22). This preemptive approach frees up the competitive slot for the circuit of $i$. As a result, users with a higher token balance have more chance to claim a competitive slot during periods of congestion, compared to free-riders, and experience higher and more stable download speeds. We consider the analysis of different allocation policies, e.g., using a packet-granular scheduler [56], as further work.

### 7.3. Data collection

We integrate both ConTrib and the slot assignment logic in Tribler and release a new version of our software. We also deploy a crawler that continuously requests ConTrib records from random peers in the Tribler network. This crawler selects a random peer in the ConTrib network every two seconds and requests missing records in their personal ledger. The crawler statistics are published on a public website.[6] A deployment period of two years has resulted in more than 160 million records, created by over 94’000 peers. The crawler stores collected records in a sqlite database that is enhanced with additional indices to speed up insertion and analysis queries. The file size of the database with all collected records is around 120 GB, and we plan on releasing the full data set later. Our crawler discovered 127’135 proofs of fraud. We also find that 11.4% of all collected proposals in the deployed ConTrib network is unconfirmed. This fraction of unconfirmed proposals is either because the proposal counterparty has not created a confirmation, or because our crawler has not picked up the confirmation record.

Deploying a crawler and monitoring the records created by ConTrib allows us to detect anomalies caused by software bugs or unexpected user behaviour. It also provides us with the means to monitor the growth of users within Tribler by tracking the number of unique peers in the ConTrib network. We have also included a creation timestamp in the payload of each record created with Tribler, allowing us to perform a time-based analysis.

Fig. 13 shows the daily number of created proposal records. We annotate the dates on which we released a major version of Tribler. Fig. 13 reveals that more users run Tribler during the weekend and create more proposals on a Saturday and a Sunday. We also observed two large-scale outage of exit nodes, in April 2019 and May 2020, likely due to infrastructure failures. Despite these outages, users would still perform payouts when downloading directly from other Tribler users without anonymity.

### 7.4. Free-rider identification and service refusal

To show the effect of ConTrib and our slot assignment mechanism on free riders, we deploy 48 exit nodes in the Tribler network, running on the same machine. Each exit node is configured with a total of 10 random slots and 20 competitive slots, resulting in a total of 1440 slots. We determined this number of random and competitive slots based on the hardware capacity of our machine. We are specifically interested in the situation when a peer is unable to claim a slot, due to their bandwidth token balance being insufficient. We refer to this situation as a *reject event*. For each reject event, we log the bandwidth token balance of the rejected peer. In total, we logged over 1.4 million reject events over three weeks.

Fig. 14 shows an Empirical Cumulative Distribution Function (ECDF) with the bandwidth token balances of all peers (dotted green line) and the balances associated with rejected circuit requests (solid red line). For presentation clarity, we filter out all peers and reject events with balances higher than 50 GB or lower than −500 GB. Many Tribler users have a negative bandwidth token balance. The median token balance of all users is −713 MB. This number suggests that there is not much opportunity to earn bandwidth tokens by contributing to the network. By default, the Tribler software downloads content over a 1-hop circuit, only involving an exit node. Changing the default behaviour to use 2-hop downloads could alleviate this issue, at the cost of decreased download speeds. Fig. 14 also shows that users with a relatively low (e.g., $<$ 50 GB) are frequently rejected a slot. The median token balance associated with reject events is −181.4 GB, demonstrating that our mechanism effectively targets peers with lower bandwidth token balances. The slots claimed by free-riders will likely go to dedicated peers when the network is congested. This deployment trial shows that ConTrib is effective at detecting and

[6] See http://explorer.tribler.org.

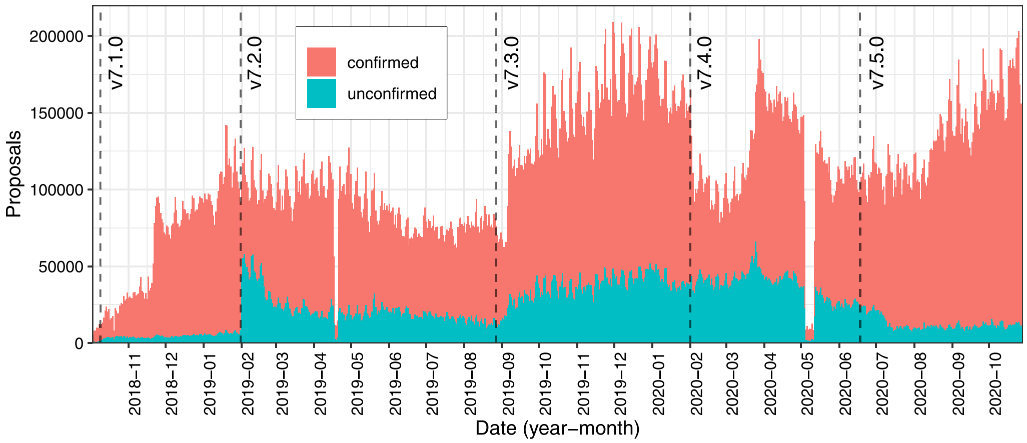


**Fig. 13.** Daily creation statistics of proposals, during our two-year deployment trial. We show the amount of confirmed and unconfirmed proposals. We annotate the major releases of Tribler.

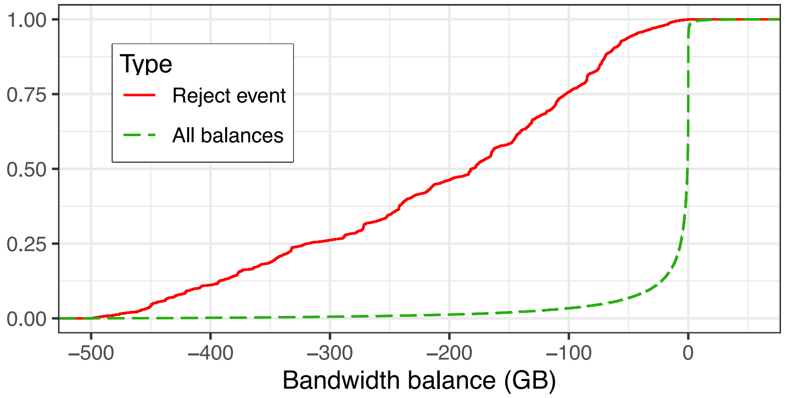


**Fig. 14.** Empirical Cumulative Distribution Function (ECDF) of the bandwidth token balances of peers and individual rejects events at exit nodes.

addressing free-riding behaviour in Tribler. The integration of ConTrib has increases network performance and helps to maintain fairness amongst downloading users.

## 8. Conclusion

We have presented ConTrib, a universal accounting mechanism to maintain fairness in decentralized applications by accounting work. The ConTrib data structure uses records to capture the work performed by peers. Each peer maintains a tamper-evident personal ledger with interlinked records. Fraud, the illegitimate modification of a record in ones personal ledger, is optimistically detected through the random exchange of records and thorough validation of incoming ones. We have devised a system architecture of ConTrib and implemented it. Our evaluation has demonstrated that ConTrib is capable of detecting fraud within seconds, even when the network grows to 10’000 peers and when scaling the system load. Through a two-year deployment trial of ConTrib in Tribler, involving more than 94’000 users, we have successfully addressed free-riding behaviour.

We envision and encourage the usage of ConTrib beyond work accounting in decentralized applications. Currently, ConTrib is being evaluated in different scenarios that require accountability, including decentralized trading, software developer portfolios, and self-sovereign identity [43,51,52].

## CRediT authorship contribution statement

**Martijn de Vos:** Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing - original draft, Writing - review & editing, Visualization. **Johan Pouwelse:** Conceptualization, Resources, Visualization, Supervision, Project administration, Funding acquisition.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## References

[1] J. Frost, L. Gambacorta, Y. Huang, H.S. Shin, P. Zbinden, Bigtech and the changing structure of financial intermediation, Econ. Policy (2019).

[2] M.L. Best, The internet that facebook built, Commun. ACM 57 (12) (2014) 21–23.

[3] J. Schor, et al., Debating the sharing economy, J. Self-Gov. Manage. Econ. 4 (3) (2016) 7–22.

[4] K. Frenken, J. Schor, Putting the sharing economy into perspective, in: A Research Agenda for Sustainable Consumption Governance, Edward Elgar Publishing, 2019.

[5] E. Bokányi, A. Hannák, Understanding inequalities in ride-hailing services through simulations, Sci. Rep. 10 (1) (2020) 6500, http://dx.doi.org/10.1038/s41598-020-63171-9, URL http://www.nature.com/articles/s41598-020-63171-9.

[6] B. Kotapati, S. Mutungi, M. Newham, J. Schroeder, S. Shao, M. Wang, The antitrust case against apple, 2020, Available at SSRN.

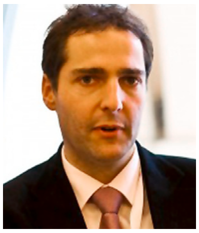
[7] S. Nakamoto, Bitcoin: A Peer-To-Peer Electronic Cash System, Tech. Rep., Manubot, 2019.

[8] I. Bashir, Mastering Blockchain: Distributed Ledger Technology, Decentralization, and Smart Contracts Explained, Packt Publishing Ltd, 2018.

[9] G. Wood, et al., Ethereum: A secure decentralised generalised transaction ledger, Ethereum Proj. Yellow Pap. 151 (2014) (2014) 1–32.

[10] R. Dingledine, N. Mathewson, P. Syverson, Tor: The Second-Generation Onion Router, Tech. Rep., Naval Research Lab Washington DC, 2004.
[11] T. Locher, et al., Free riding in bittorrent is cheap, in: HotNets, 2006.
[12] M. Sirivianos, J.H. Park, R. Chen, X. Yang, Free-riding in bittorrent networks with the large view exploit, in: IPTPS, 2007.
[13] M. Jelasity, A. Montresor, O. Babaoglu, Detection and Removal of Malicious Peers in Gossip-Based Protocols, szte, 2004.
[14] J.A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D.H. Epema, M. Reinders, M.R. Van Steen, H.J. Sips, Tribler: a social-based peer-to-peer system, Concurr. Comput. Pract. Exp. 20 (2) (2008) 127–138.
[15] M. de Vos, J. Pouwelse, A blockchain-based micro-economy of bandwidth tokens, in: CompSys 2018, 2018.
[16] R.T. Ma, et al., An incentive mechanism for P2p networks, in: ICDCS, IEEE, 2004, pp. 516–523.
[17] G. Ruffo, R. Schifanella, Fairpeers: Efficient profit sharing in fair peer-to-peer market places, J. Netw. Syst. Manage. 15 (3) (2007) 355–382.
[18] D.P. Anderson, Boinc: A platform for volunteer computing, J. Grid Comput. (2019) 1–24.
[19] D. Easley, M. O'Hara, S. Basu, From mining to markets: The evolution of bitcoin transaction fees, J. Financ. Econ. 134 (1) (2019) 91–109.
[20] J. Benet, N. Greco, Filecoin: A Decentralized Storage Network, Tech. Rep., Protoc. Labs, 2018, pp. 1–36.
[21] M. Ghosh, et al., A TorPath to TorCoin: Proof-of-Bandwidth Altcoins for Compensating Relays, Tech. Rep., Naval Research, 2014.
[22] T. Hummel, O. Stramme, R.M. La Salle, Earning a living among peers-the quest for viable p2p revenue models, in: 36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the, IEEE, 2003, pp. 10–pp.
[23] M. Meulpolder, et al., Bartercast: A practical approach to prevent lazy freeriding in p2p networks, in: IPDPS, IEEE, 2009, pp. 1–8.
[24] M. Karakaya, I. Korpeoglu, Ö. Ulusoy, Free riding in peer-to-peer networks, IEEE Internet Comput. 13 (2) (2009) 92–98.
[25] F. Hendrikx, K. Bubendorfer, R. Chard, Reputation systems: A survey and taxonomy, J. Parallel Distrib. Comput. 75 (2015) 184–197.
[26] E. Bellini, Y. Iraqi, E. Damiani, Blockchain-based distributed trust and reputation management systems: A survey, IEEE Access 8 (2020) 21127–21151.
[27] B. Cohen, Incentives build robustness in bittorrent, in: Workshop on Economics of Peer-To-Peer Systems, Vol. 6, Berkeley, CA, USA, 2003, pp. 68–72.
[28] J. Benet, Ipfs-content addressed, versioned, p2p file system, 2014, arXiv preprint arXiv:1407.3561.
[29] D.S. Wallach, P. Druschel, et al., Enforcing fair sharing of peer-to-peer resources, in: International Workshop on Peer-To-Peer Systems, Springer, 2003, pp. 149–159.
[30] I. Osipkov, P. Wang, N. Hopper, Robust accounting in decentralized P2p storage systems, in: 26th IEEE International Conference on Distributed Computing Systems (ICDCS'06), IEEE, 2006, p. 14.
[31] R. Guerraoui, et al., Lifting: lightweight freerider-tracking in gossip, in: Middleware, Springer, 2010, pp. 313–333.
[32] S.B. Mokhtar, J. Decouchant, V. Quéma, Acting: Accurate freerider tracking in gossip, in: 2014 IEEE 33rd International Symposium on Reliable Distributed Systems, IEEE, 2014, pp. 291–300.
[33] S. Seuken, D.C. Parkes, Sybil-proof accounting mechanisms with transitive trust, in: Proceedings of the International Foundation for Autonomous Agents and Multiagent Systems, ACM, 2014.
[34] A. Haeberlen, et al., Peerreview: Practical accountability for distributed systems, SIGOPS 41 (6) (2007) 175–188.
[35] A. Diarra, et al., Fullreview: Practical accountability in presence of selfish nodes, in: SRDS, IEEE, 2014, pp. 271–280.
[36] P. Otte, et al., Trustchain: A sybil-resistant scalable blockchain, Future Gener. Comput. Syst. 107 (2020) 770–780.
[37] S. Crosby, et al., Efficient data structures for tamper-evident logging, in: USENIX Security, 2009, pp. 317–334.
[38] D. Hausheer, B. Stiller, Peermint: Decentralized and secure accounting for peer-to-peer applications, in: International Conference on Research in Networking, Springer, 2005, pp. 40–52.
[39] J.R. Douceur, The sybil attack, in: International Workshop on Peer-To-Peer Systems, Springer, 2002, pp. 251–260.
[40] F. Li, P. Mittal, M. Caesar, N. Borisov, SybilControl: Practical Sybil defense with computational puzzles, in: Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing, 2012, pp. 67–78.
[41] R. Delaviz, N. Andrade, J.A. Pouwelse, D.H. Epema, Sybilres: A sybil-resilient flow-based decentralized reputation mechanism, in: 2012 IEEE 32nd International Conference on Distributed Computing Systems, IEEE, 2012, pp. 203–213.
[42] H. Yu, P.B. Gibbons, M. Kaminsky, F. Xiao, Sybillimit: A near-optimal social network defense against sybil attacks, in: 2008 IEEE Symposium on Security and Privacy (Sp 2008), IEEE, 2008, pp. 3–17.
[43] Q. Stokkink, J. Pouwelse, Deployment of a blockchain-based self-sovereign identity, in: 2018 IEEE International Conference on Internet of Things (IThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), IEEE, 2018, pp. 1336–1342.
[44] A. Stannat, C. Umut Ileri, D. Gijswijt, J. Pouwelse, Achieving sybil-proofness in distributed work systems, in: International Conference on Autonomous Agents and Multiagent Systems, 2021.
[45] Y. Liu, Y. Yang, Y.L. Sun, Detection of collusion behaviors in online reputation systems, in: 2008 42nd Asilomar Conference on Signals, Systems and Computers, IEEE, 2008, pp. 1368–1372.
[46] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, G. Maxwell, Bulletproofs: Short proofs for confidential transactions and more, in: 2018 IEEE Symposium on Security and Privacy (SP), IEEE, 2018, pp. 315–334.
[47] A. Poelstra, A. Back, M. Friedenbach, G. Maxwell, P. Wuille, Confidential assets, in: International Conference on Financial Cryptography and Data Security, Springer, 2018, pp. 43–63.
[48] R. Krishnan, M. Smith, Z. Tang, R. Telang, The virtual commons: Why free-riding can be tolerated in file sharing networks, in: ICIS 2002 Proceedings, 2002, p. 82.
[49] M. Vukolić, The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication, in: INetSec, Springer, 2015, pp. 112–125.
[50] I. Osipkov, E.Y. Vasserman, N. Hopper, Y. Kim, Combating double-spending using cooperative P2p systems, in: 27th International Conference on Distributed Computing Systems (ICDCS'07), IEEE, 2007, p. 41.
[51] M. de Vos, C.U. Ileri, J. Pouwelse, Xchange: A universal mechanism for asset exchange between permissioned blockchains, World Wide Web (2021) 1–38.
[52] M. de Vos, M. Olsthoorn, J. Pouwelse, Devid: Blockchain-based portfolios for software developers, in: 2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON), IEEE, 2019, pp. 158–163.
[53] M. de Vos, J. Pouwelse, Real-time money routing by trusting strangers with your funds, in: 2018 IFIP Networking Conference (IFIP Networking) and Workshops, IEEE, 2018, pp. 1–9.
[54] N. Matloff, Introduction to Discrete-Event Simulation and the Simpy Language, Dept of Computer Science. University of California at Davis. Retrieved on August, Davis, CA, 2008, pp. 1–33, Retrieved on August 2 (2009).
[55] P. Palmieri, J. Pouwelse, Paying the guard: an entry-guard-based payment system for tor, in: FC, Springer, 2015, pp. 437–444.
[56] R. Jansen, M. Traudt, J. Geddes, C. Wacek, M. Sherr, P. Syverson, Kist: Kernel-informed socket transport for tor, ACM Trans. Priv. Secur. 22 (1) (2018) 1–37.

**Martijn de Vos** is a Ph.D. student at Delft University of Technology and member of the Delft Blockchain Lab. His research interest include accountability protocols, peer-to-peer e-commerce, decentralized marketplaces, and blockchain technology. He is also a developer of the Tribler file-sharing application, installed by 1.8 million people over the past decade.



**Johan Pouwelse** is an associate professor at Delft University of Technology, specialized in large-scale cooperative systems. During his Ph.D. he created the first system for cooperative resource management for portable devices on Linux. This resulted in the first portable hardware and Linux driver capable of reducing CPU frequency and voltage from user-space in 2001. The driver got accepted into the Linux kernel and this architecture is still used by every Android and iOS device. Also, he conducted the first resource usage measurements for IEEE 802.11b, known know as WiFi. After receiving his Ph.D., he conducted one of the largest measurements of the BitTorrent P2P network. He founded the Tribler video-on-demand client in 2005, it has been installed by 1.8 million people over the past decade. Tribler serves as a living laboratory and proving ground for next generation self-organizing systems research and ledger technology. In 2007 his research team launched BarterCast, a primitive distributed ledger. Dr. Pouwelse has (co-)authored over 129 scientific papers.