Persistent BitTorrent Trackers

François-Xavier Wicht
University of Bern

Zhengwei Tong
Duke University

Shunfan Zhou
Phala Network

Hang Yin
Phala Network

Aviv Yaish
Yale University, IC3

Abstract—**Private BitTorrent trackers enforce upload-to-download ratios to prevent free-riding, but suffer from three critical weaknesses: reputation cannot move between trackers, centralized servers create single points of failure, and upload statistics are self-reported and unverifiable. When a tracker shuts down (whether by operator choice, technical failure, or legal action) users lose their contribution history and cannot prove their standing to new communities. We address these problems by storing reputation in smart contracts and replacing self-reports with cryptographic attestations. Receiving peers sign receipts for transferred pieces, which the tracker aggregates and verifies before updating on-chain reputation. Trackers run in Trusted Execution Environments (TEEs) to guarantee correct aggregation and prevent manipulation of state. If a tracker is unavailable, peers use an authenticated Distributed Hash Table (DHT) for discovery: the on-chain reputation acts as a Public Key Infrastructure (PKI), so peers can verify each other and maintain access control without the tracker. This design persists reputation across tracker failures and makes it portable to new instances through single-hop migration in factory-deployed contracts. We formalize the security requirements, prove correctness under standard cryptographic assumptions, and evaluate a prototype on Intel TDX. Measurements show that transfer receipts adds less than 6% overhead with typical piece sizes, and signature aggregation speeds up verification by $2.5\times$.**

Index Terms—**p2p, file transfer, censorship resistance, reputation-based system, distributed file exchange.**

1. Introduction

BitTorrent has become one of the most widely deployed peer-to-peer (p2p) protocols, responsible for a significant fraction of global internet traffic. While public BitTorrent trackers allow unrestricted participation, they suffer from endemic free-riding: users who download content without contributing equivalent uploads [1]. Private[1] trackers emerged as a community-driven solution, restricting access to users who maintain favorable upload-to-download ratios. These systems store reputation in centralized databases, creating vibrant but fragile communities vulnerable to single points of failure. This design hampers quick recovery from failures, as illustrated by a notable incident. In 2007, European police agencies shut down the prominent OiNK tracker, which was called "the world's greatest record store" [36], with its founder even named one of online music's most influential people [14]. While alternative trackers were rapidly created, admissions were often invitation-only and ex-OiNK members could not migrate their hard-earned ratios [18].

The private tracker model exhibits three critical structural weaknesses. *First*, upload-to-download ratios are not portable across trackers. Communities operate as isolated silos, preventing users from leveraging their contribution history when joining new trackers or recovering from shutdowns. *Second*, centralization creates fragility: trackers serve as single points of failure for both reputation storage and peer discovery. While distributed solutions like DHTs [28] can handle peer discovery, they lack authentication and are thus disabled for private tracker torrents [4]. *Third*, transfer statistics are self-reported and unverifiable. Users can inflate ratios through false reports [6], with only ex post moderation available to detect fraud.

1.1. Our work

We redesign the private tracker architecture to eliminate these three weaknesses through blockchain-based reputation and cryptographic attestation.

First, we persist reputation and make it portable through smart contracts that record user contributions on-chain. Users can migrate reputation to new trackers, join federated communities, or bootstrap new tracker instances. When a tracker shuts down, no reputation is lost: the blockchain preserves all historical contributions, enabling seamless migration to successor communities.

Second, we eliminate both forms of centralization. For reputation storage, smart contracts replace centralized databases, thus no single entity controls or can destroy reputation data. The tracker posts cryptographically authenticated state transitions to the blockchain for accountability. Should a tracker fail or become compromised, the contract enables rollback to the last consistent state. For peer discovery, on-chain reputation serves as an authenticated allow-list, letting peers to fall back on DHT-based discovery when trackers are down. This decentralized fallback maintains access control and eliminates the single point of failure risk posed by the tracker in peer discovery.

Third, we introduce an attestation protocol where peers cryptographically sign evidence of data transfers. The tracker aggregates attestations, creating an auditable chain of custody for reported statistics. Senders cannot inflate contributions without obtaining receivers' signatures, and disputes can be resolved by examining cryptographic receipts rather than relying on ex post moderation.

---

[1] Following BitTorrent's standard terminology, "private" does not refer to privacy notions such as anonymity or unlinkability, but rather to permissioned resources (as opposed to publicly accessible ones).

The result is a *persistent tracker*: a censorship-resistant protocol for private content distribution that aligns with Web3 principles of decentralization, verifiability, and user sovereignty. We formalize the security requirements, present a construction with per-piece attestation, and demonstrate how blockchain-based persistent storage combined with cryptographic mechanisms achieves robust guarantees even against powerful adversaries.

Our contributions are as follows:

1. A formal *Persistent BitTorrent Tracker Scheme* (PBTS) with algorithms, security requirements, and proofs under standard cryptographic assumptions.
2. A construction adding verifiable per-piece attestation to BitTorrent, replacing self-reported statistics with cryptographically checked transfers.
3. Several optimizations for attestation to reduce signing cost, bandwidth, and verification overhead.
4. A portable reputation system using factory-based smart contracts where new trackers inherit state from predecessors through single-hop migration.
5. An authenticated DHT fallback using on-chain reputation as PKI, maintaining access control when trackers are unavailable.
6. An implementation and evaluation on Intel TDX hardware, demonstrating that PBTS achieves strong security guarantees with less than 6% throughput overhead for typical workloads and scalable verification via signature aggregation.

The remainder of this paper is structured as follows. Section 2 surveys related work on file-sharing incentives, on-chain reputation systems, persistent distributed systems and usages of Trusted Execution Environments (TEEs). Section 3 provides background on the BitTorrent protocol and establishes the needed notation and cryptographic primitives. Section 4 formally defines the Persistent BitTorrent Tracker Scheme, presents our construction with per-piece attestation, factory-based smart contracts for portable reputation, and authenticated DHT fallback for tracker-less operation. Section 5 analyzes the security properties of the construction. Section 6 describes implementation details. Section 7 evaluates the prototype.

2. Related work

Fairness in File-sharing. Ensuring fairness is a long-standing challenge in open p2p systems, e.g., preventing "free-riding" by users [1], that is, users who only download content without uploading at least an equivalent amount to others. Some have proposed mitigating this issue by requiring micro-payments for downloads, possibly by protocol-specific currencies [19]. BitTorrent attempts to address this via an optional "choking" protocol where a user may temporarily refuse to upload to peers who do not reciprocate [13]. The analysis of Wu and Zhang [37] shows that such tit-for-tat protocols quickly converge to an efficient equilibrium: bandwidth is optimally allocated.

Private Trackers. Private trackers emerged as a community-driven solution to free-riding, introducing admission-control and a reputation layer based on upload-to-download ratios to enforce sharing norms [23]. Thus, communities typically only admit new users with a good upload-to-download ratio in other communities, and kick out existing users who do not maintain a good ratio. The study of Hales et al. [21] identifies potential "credit squeezes" where a lack of upload opportunities can stifle participation in private trackers. While some propose to improve fairness by applying economic inequality measures (e.g., the Gini coefficient) to file-sharing communities [32], recent work finds that such measures are inaccurate in pseudonymous settings [38].

Sybil Attacks in BitTorrent. A notable issue that may arise in BitTorrent communities is that actors can fake their upload-to-download ratio, whether by falsely reporting uploads [6], or by creating multiple identities who upload and download from each other. The latter manipulation is part of a broader class of so-called *Sybil* attacks that involve the creation of "fake" identities [15]. While BitTorrent's tit-for-tat is resistant to some manipulations [12], it is vulnerable to Sybil attacks [25]. Cheng, Deng, and Li [10] show that the gain that can be obtained by such attacks equals at most three times the amount of data that could be downloaded honestly, with a tight bound of two obtained later by Cheng et al. [11]. Prior work analyzes the economic impact of file-sharing services on a market [29], and how such services should be priced [26], with later work showing that when incentives are not correctly aligned, Sybil attacks may drain victim resources [20].

Rethinking BitTorrent's Incentives. Market-based solutions for BitTorrent's vulnerability to Sybil attacks and other manipulations have been explored by prior work. For example, Levin et al. [25] provide an elegant analogy: from the perspective of a user, the peers competing for its upload bandwidth are participating in an auction. The authors find that allocating upload bandwidth proportionally to the incoming bandwidth received from each peer is nearly an equilibrium. That is, deviating from this strategy is nearly unprofitable, given that all other peers are following the rules. A different design is offered by Zohar and Rosenschein [40], where, by default, peers cannot request specific data blocks, but rather a range from which data blocks are chosen at random. A user can make specific requests to a peer only in exchange for fully providing blocks asked for by that peer.

Reputation Management. A notable line of work proposed systems to persist and manage reputation. One prominent design for such systems is based on the non-transferable "soulbound tokens" of Ohlhaver, Weyl, and Buterin [30], which can be used to represent identity, and, by extension, reputation. The authors emphasize the importance of having a recovery mechanism in place, to assist those who for whatever reason lost control of their tokens (e.g., due to losing the secret key corresponding to the account holding the tokens). We highlight another crucial recovery notion, for the case where the system itself is compromised. Alternative reputation management systems similarly lack recovery functionality of this sort, such as the UniRep protocol [35]. A general ZKP-based design offering functionality similar to UniRep's is provided by

Buterin [7]. A framework called zk-promises is put forth by Shih et al. [34], which re-purposes methods used by privacy-preserving cryptocurrency protocols to endow private reputation systems with moderation capabilities (e.g., to enable blacklisting accounts).

Persistent Systems. A main goal of ours is to provide takedown resistance for BitTorrent trackers. Previous work did not consider this threat model, mostly focusing on network-level interference. For example, Bocovich et al. [3] devise an internet-censorship circumvention system which relies on rapidly setting up numerous temporary proxies, which, due to their sheer number, are harder to block. To reduce the costs associated with launching such proxies, Kon et al. [24] present a service called SpotProxy which continuously searches for cheap hosting providers, and, if indeed found, deploys new proxy instances and migrates clients to them. In comparison to our work, SpotProxy relies on a central controller to save client registration details and handle migration.

TEEs. Previous work employed TEEs to design robust systems in other settings and with different objectives than ours. A line of work started by Zhang et al. [39] takes this approach to design authenticated data-feeds for smart contracts. In the context of identity, Maram et al. [27] employ such data-feeds to scrape credentials from websites and transfer them on-chain in a trustworthy manner, and to prevent Sybils, MPC is used to deduplicate imported credentials. In another line of work, Druschel and Rowstron [16] use trusted hardware for persistent storage, achieved by replicating files across multiple nodes.

3. Preliminaries

Notations. We write $x \leftarrow\!\!\$\; S$ to denote sampling x uniformly at random from set S. A function $\nu : \mathbb{N} \to \mathbb{R}$ is *negligible* if for every polynomial $p(\cdot)$, there exists $N \in \mathbb{N}$ such that for all $n > N$, $|\nu(n)| < 1/p(n)$. We write $[n]$ to denote the set $\{1, \ldots, n\}$.

We define reputation as a function $\mathsf{Rep} : \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \to \mathbb{R}$ mapping an account's total uploaded and downloaded data to a numerical score. A common instantiation is the sharing ratio $\rho = \frac{\text{uploaded}}{\text{downloaded}}$ (with $\rho = \infty$ when downloaded $= 0$), though trackers may implement alternative reputation functions (e.g., crediting upload contributions more heavily, or incorporating time-weighted statistics).

BitTorrent protocol and tracker architecture. BitTorrent is a p2p protocol for file distribution. A file is divided into fixed-size pieces, which peers exchange until the full content is reconstructed. Clients advertise which pieces they hold and prioritize peers who reciprocate, enforcing tit-for-tat incentives [13]. Each torrent is described by a `.torrent` file containing metadata, including the file length, piece hashes, and tracker URLs. Piece hashes guarantee content integrity, while the tracker maps torrent identifiers (infohashes) to active peers. When queried, it returns a random subset of peers for the client to contact. Once peers discover each other, they exchange handshakes and begin transferring data. Each peer decides locally whom to upload to, based on choking heuristics. *Public* trackers permit unrestricted access, while *private* trackers bind user accounts to credentials and enforce upload/download quotas [8]. Account accumulate reputation, typically a download-to-upload ratio. Clients periodically report their statistics to the tracker, which stores them in a centralized database. These reports are unauthenticated and rely on client honesty. Moderators may intervene to detect fraud, but audits are manual and retrospective.

Cryptographic primitives. Our construction relies on three building blocks: aggregatable signatures for efficient batching of peer attestations, smart contracts for persistent on-chain reputation storage, and trusted execution environments for authenticated tracker operation. We formalize each primitive below.

Definition 3.1 (Aggregatable signature scheme)**.** *An aggregatable signature scheme Σ consists of five algorithms:*

- $\mathsf{KeyGen}(1^\lambda) \to (\mathsf{sk}, \mathsf{pk})$*: Takes a security parameter λ and outputs a key pair consisting of a secret signing key* sk *and a public verification key* pk.
- $\mathsf{Sign}(\mathsf{sk}, m) \to \sigma$*: Takes a secret key* sk *and a message $m \in \{0,1\}^*$, and outputs a signature σ.*
- $\mathsf{Verify}(\mathsf{pk}, m, \sigma) \to \{0,1\}$*: Takes a public key* pk*, a message m, and a signature σ, and outputs* 1 *if the signature is valid, or* 0 *otherwise.*
- $\mathsf{Agg}(\{(\mathsf{pk}_i, m_i, \sigma_i)\}_{i\in[n]}) \to \sigma_{\mathsf{agg}}$*: Takes a set of public keys, messages, and signatures, and outputs an aggregate signature σ_{agg}.*
- $\mathsf{AggVer}(\{(\mathsf{pk}_i, m_i)\}_{i\in[n]}, \sigma_{\mathsf{agg}}) \to \{0,1\}$*: Takes a set of public key-message pairs and an aggregate signature, and outputs* 1 *if the aggregate signature is valid for all pairs, or* 0 *otherwise.*

The signature scheme is required to satisfy correctness*: for all* $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$ *and all messages m, then* $\mathsf{Verify}(\mathsf{pk}, m, \mathsf{Sign}(\mathsf{sk}, m)) = 1$*. For aggregate signatures, if each signature $\sigma_i = \mathsf{Sign}(\mathsf{sk}_i, m_i)$ is valid, then:* $\mathsf{AggVer}(\{(\mathsf{pk}_i, m_i)\}_{i\in[n]}, \mathsf{Agg}(\{(\mathsf{pk}_i, m_i, \sigma_i)\}_{i\in[n]})) = 1$.

We also require existential unforgeability under chosen message attacks *(EUF-CMA): no probabilistic polynomial-time adversary, given* pk *and access to a signing oracle* $\mathsf{Sign}(\mathsf{sk}, \cdot)$*, can produce a valid signature σ^* on a message m^* that was not queried to the oracle, except with negligible probability. For aggregatable signatures, we additionally require* aggregate unforgeability*: an adversary with access to multiple signing oracles and the ability to observe aggregate signatures cannot forge a valid aggregate signature for a set of messages that includes at least one message not queried to any oracle.*

Standard instantiations include BLS signatures over pairing-friendly elliptic curves (e.g., BLS12-381), which support efficient signature aggregation. For non-aggregatable operations, ECDSA (e.g., secp256k1) and EdDSA variants (e.g., Ed25519) can also be used.

Smart contracts serve as the persistent storage layer for reputation data, replacing centralized databases with blockchain-based state that survives tracker failures. We model smart contracts as programs with authenticated write access and public read access.

Definition 3.2 (Smart contract)**.** *A smart contract is a blockchain-deployed deterministic program that maintains persistent state and is invoked by transactions. Our construction requires the following contract operations:*

- $\mathsf{SC.Init}(\mathsf{params}) \to \mathsf{addr}$*: Deploys a new contract with initialization parameters* params *and returns its on-chain address* addr.
- $\mathsf{SC.Read}(\mathsf{addr}, \mathsf{key}) \to \mathsf{value}$*: Reads the value associated with key* key *from the contract at address* addr*. This operation is publicly accessible and does not modify the contract's state.*
- $\mathsf{SC.Write}(\mathsf{addr}, \mathsf{key}, \mathsf{value}, \mathsf{auth}) \to \{\mathsf{success}, \perp\}$*: Writes* value *to the contract at address* addr *under key* key*, authenticated by* auth*. The contract enforces access control: only authorized entities (e.g., the tracker's TEE) can modify state. Returns* $\mathsf{success}$ *if the write succeeds, or* $\perp$ *if authorization fails.*

Smart contracts guarantee integrity*: state transitions are validated by consensus among blockchain nodes, ensuring that unauthorized modifications are rejected. They also provide* persistence*: once written, data remains immutable and accessible as long as the blockchain operates. In our construction, we use a factory pattern where a single factory contract deploys multiple reputation contracts, each maintaining user statistics for a tracker instance.*

Trackers must aggregate receipts correctly and update on-chain state authentically, but operators cannot be trusted. TEEs ensure that tracker code executes as specified and provide confidentiality guarantees for peer data (IP addresses, activity patterns) from tracker operators.

Definition 3.3 (Trusted execution environment)**.** *A* Trusted Execution Environment *(TEE) is a secure area within a processor that provides isolated execution for sensitive code and data. TEEs guarantee confidentiality (data is inaccessible to the host OS), integrity (code cannot be tampered with), and attestation (cryptographic proofs that specific code runs in a genuine TEE).*

Modern VM-level TEE solutions such as Intel TDX [9] and AMD SEV [2] let unmodified applications run in isolated virtual machines. The attestation mechanism lets third parties cryptographically verify any system component through measurement registers that capture build-time and runtime measurements of the executed code.

While TEEs provide strong confidentiality and integrity guarantees, they do not guarantee liveness or availability. The host system retains control over resource scheduling, TEE initialization, and system call execution.

Throughout the work, a dark background denotes code executed in a TEE. Code running in a TEE implicitly outputs a certificate of correct execution (attestation).

Threat model. TEEs (Intel TDX/AMD SEV) are trusted for isolation and attestation and the blockchain is trusted for smart contract execution. Tracker operators are *untrusted*: TEE protections prevent them from tampering with execution or reading sensitive data, but they can jeopardize liveness by terminating instances or denying resources. Peers and DHT nodes are also *untrusted*. The private tracker employs a Sybil-resistant registration mechanism, such as interviews or accountable sponsorship where sponsors are held responsible for invitees' behavior (common in private trackers like RED [33]), to limit initial account creation. While peers cannot forge receipts cryptographically, colluding peers could exchange real data to generate legitimate receipts. However, this has no net benefit: downloaders always record a reputation loss, and accountable sponsorship makes creating accounts costly since sponsors risk penalties for invitees' misconduct.

4. Persistent BitTorrent tracker system

We now introduce our Persistent BitTorrent Tracker System (PBTS), which combines authenticated tracker execution, smart contracts for persistent reputation storage, and p2p cryptographic attestation. Reputation is portable due to on-chain storage, the architecture is censorship-resistant through authenticated state updates and contract-based recovery, and transfer statistics are verifiable due to cryptographic receipts that replace self-reporting.

4.1. Formal specification

PBTS extends the traditional tracker interface with multiple algorithms. Setup and KeyGen initialize the system and generate user keys. Register creates accounts authenticated by signatures. Announce provides peer discovery with reputation-based access control. Report submits verified transfer statistics backed by aggregated cryptographic receipts. Attest and Verify implement p2p attestation, where receivers sign acknowledgments of transfers. Migrate enables reputation portability by creating new tracker instances that inherit state from predecessors.

Definition 4.1 (Persistent BitTorrent tracker scheme)**.** *A* Persistent BitTorrent Tracker Scheme *(PBTS) is a tuple of eight algorithms defined as follows:*

- $\mathsf{Setup}(1^\lambda, \mathsf{MinRep}, \mathsf{InitCredit}) \to \mathsf{pp}$*: Takes security parameter* λ*, minimum reputation threshold* MinRep*, and initial upload credit* $\mathsf{InitCredit}$ *for new users, and outputs public parameters* pp *including tracker instance ID* iid*. The public parameters* pp *are implicit input to the remaining algorithms.*
- $\mathsf{KeyGen}() \to (\mathsf{sk}, \mathsf{pk})$*: Generates user key pair.*
- $\mathsf{Register}(\mathsf{uid}, \mathsf{pk}, \sigma, \mathsf{params}) \to \{0, 1\}$*: Registers user with user ID* uid*, public key* pk*, and signature* σ *over registration message including instance ID and user ID, with optional parameters* params*. Returns* 1 *if registration succeeds,* 0 *otherwise.*
- $\mathsf{Announce}(\mathsf{uid}, \mathsf{pk}, \sigma, \mathsf{tid}, \mathsf{event}) \to \mathcal{P}$*: Announces torrent* tid *with user ID* uid*, public key* pk*, signature* σ *for authentication, and event type* $\mathsf{event} \in \{\texttt{started}, \texttt{stopped}, \texttt{completed}, \texttt{none}\}$*, returning peer list* $\mathcal{P}$*.*
- $\mathsf{Report}(\mathsf{uid}, \mathsf{pk}, \{\mathsf{pk}_j\}_{j\in\mathcal{J}}, \mathcal{T}, \{t_j\}_{j\in\mathcal{J}}, \sigma_{\mathsf{agg}}, \Delta_{\mathsf{up}}, \Delta_{\mathsf{down}}) \to \{0, 1\}$*: Reports transfer statistics with user ID* uid*, user's public key* pk*, set of peer public keys* $\{\mathsf{pk}_j\}_{j\in\mathcal{J}}$ *who provided receipts, torrent metadata* $\mathcal{T}$*, timestamps* $\{t_j\}_{j\in\mathcal{J}}$ *for each receipt, aggregated signature* σ_{agg} *for all receipts, upload delta* Δ_{up}*, and download delta* Δ_{down}*. The tracker*

reconstructs receipts with timestamps and verifies aggregate signature. Returns 1 *if accepted,* 0 *otherwise.*

- $\mathsf{Attest}(\mathsf{sk}_{\mathsf{receiver}}, \mathsf{pk}_{\mathsf{sender}}, p_i, \mathcal{T}, t_{epoch}) \rightarrow \sigma_{\mathsf{receipt}}$*: Generates cryptographic receipt for piece transfer, where receiving peer signs acknowledgment for sending peer. Takes receiver's secret key, sender's public key, piece* p_i*, torrent metadata* $\mathcal{T} = (h_{\mathcal{T}}, [h_1, \ldots, h_n])$ *containing infohash and piece hashes, and epoch timestamp* t_{epoch}*. Returns receipt signature binding infohash, sender's public key, piece hash, piece index, and epoch.*
- $\mathsf{Verify}(\mathsf{pk}_{\mathsf{receiver}}, \mathsf{pk}_{\mathsf{sender}}, p_i, \mathcal{T}, t_{epoch}, \sigma_{\mathsf{receipt}}) \rightarrow \{0, 1\}$*: Verifies cryptographic receipt by checking receiver's signature and piece integrity against* $\mathcal{T}$*. Returns* 1 *if valid and* 0 *otherwise. Valid receipts prove transfers from sender to receiver. Peers use this algorithm with piece data* p_i *to verify receipts locally, while the tracker only needs the piece hash* h_i *and index* i *from* $\mathcal{T}$ *for signature verification.*
- $\mathsf{Migrate}(\mathsf{addr}_{\mathsf{old}}, \pi) \rightarrow \mathsf{addr}_{\mathsf{new}}$*: Migrates reputation from old contract address* $\mathsf{addr}_{\mathsf{old}}$ *using migration proof* π*, returning new contract address* $\mathsf{addr}_{\mathsf{new}}$.

4.2. Construction

We now instantiate the PBTS scheme using BLS signatures for receipt aggregation, smart contracts for reputation storage, and TEE-based tracker execution. Figure 1 shows the system architecture.

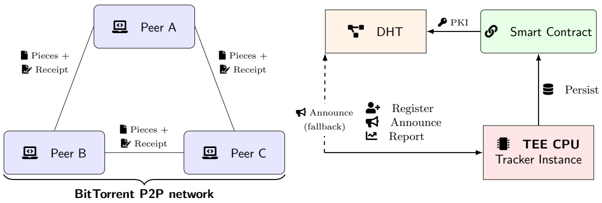

Figure 1. Architecture of the censorship-resistant tracker system. Peers (A, B, C) form a distributed swarm and exchange file pieces directly using the standard BitTorrent protocol. During file transfer, sending and receiving peers exchange cryptographic receipts. Each peer interacts with a tracker instance that ensures correct protocol execution and protects sensitive data. Peers register by proving public key ownership, announce torrents to retrieve peer lists with reputation-based access control, and report upload/download statistics with aggregated receipts. All tracker operations (*register*, *announce*, and *report*) execute securely with authenticated state updates. The tracker periodically writes reputation data to a smart contract on the blockchain, providing persistent and verifiable reputation storage. Cryptographic attestation ensures that only legitimate tracker instances can modify on-chain reputation.

Initialization. Trackers are initialized by running Setup to generate unique instance identifiers and system parameters, while users generate key pairs via KeyGen for authentication and signing, as formalized in Fig. 2.

User registration. Users register by proving ownership of their public key through a signature over a registration message containing the atom `register`, their user ID, and the tracker instance ID. The tracker verifies the signature and checks that the user ID is not already registered by reading from the smart contract. If the user ID already exists, registration is rejected to prevent duplicate accounts. Otherwise, the tracker writes the user's ID, public key, and initial reputation counters to the contract. New users receive an initial upload credit $\mathsf{InitCredit}$ to bootstrap participation, with zero downloads. Registration succeeds only if the contract write operation succeeds. Subsequent operations authenticate users via signatures over operation-specific messages using the registered public key. Cryptographic attestation ensures that only legitimate tracker instances can modify reputation data. The registration algorithm is specified in Fig. 3.

Setup (in TEE)

```
Setup($1^\lambda$, MinRep, InitCredit):
  iid $\leftarrow\!\!\$\ \{0,1\}^\lambda$
  pp $\leftarrow (\lambda, \mathsf{iid}, \mathsf{MinRep}, \mathsf{InitCredit})$
  return pp
```

Setup and key generation

```
KeyGen():
  $(\mathsf{sk}, \mathsf{pk}) \leftarrow \Sigma.\mathsf{KeyGen}(1^\lambda)$
  return $(\mathsf{sk}, \mathsf{pk})$
```

Figure 2. PBTS initialization. Setup generates tracker instance ID and system parameters (MinRep, $\mathsf{InitCredit}$). KeyGen generates key pairs for an aggregatable signature scheme.

User registration (in TEE)

```
Register(uid, pk, $\sigma$, params):
  $m \leftarrow (\mathtt{register} \,\|\, \mathsf{iid} \,\|\, \mathsf{uid})$
  if $\Sigma.\mathsf{Verify}(\mathsf{pk}, m, \sigma) = 0$ then return 0
  existing $\leftarrow \mathsf{SC.Read}(\mathsf{addr}, \mathsf{uid})$
  if existing $\neq \bot$ then return 0
  result $\leftarrow \mathsf{SC.Write}(\mathsf{addr}, \mathsf{pk}, (\mathsf{uid}, \mathsf{InitCredit}, 0), \mathsf{auth}_{\mathsf{TEE}})$
  if result $= \bot$ then return 0
  return 1
```

Figure 3. User registration with signature verification and on-chain state initialization. The tracker verifies ownership of the public key and writes user credentials to the smart contract with initial reputation counters.

Torrent announcement and peer discovery. When a peer wishes to participate in a torrent, it announces to the tracker using the $\mathsf{Announce}$ algorithm (Fig. 4). The tracker maintains internal state $\mathcal{S}_{\mathsf{tid}}$ for each torrent, storing the set of active peers. The tracker first verifies the peer's signature, then reads the peer's upload and download statistics from the smart contract using their user ID and computes their reputation score. If the peer is starting a new download and their reputation falls below the minimum threshold, access is denied. Otherwise, the tracker updates its internal swarm state: removing the peer if they are stopping, or adding their IP address and port if they are joining or continuing. The tracker then samples a random subset of active peers uniformly at random from the swarm and returns this list to the announcing peer.

P2P attestation. Traditional trackers accept self-reported statistics. We replace this with cryptographic receipts where downloaders sign acknowledgments for received

Torrent announcement (in TEE)

```
Announce(uid, pk, σ, tid, event):
  $m \leftarrow (\texttt{announce} \,\|\, \mathsf{uid} \,\|\, \mathsf{tid} \,\|\, \mathsf{event})$
  if $\Sigma.\mathsf{Verify}(\mathsf{pk}, m, \sigma) = 0$ then return $\emptyset$
  $(\mathsf{up}, \mathsf{down}) \leftarrow \mathsf{SC.Read}(\mathsf{addr}, \mathsf{uid})$
  $r \leftarrow \mathsf{Rep}(\mathsf{up}, \mathsf{down})$
  if $\mathsf{event} = \texttt{started} \land r < \mathsf{MinRep}$ then return $\emptyset$
  if $\mathsf{event} = \texttt{stopped}$ then $\mathcal{S}_{\mathsf{tid}} \leftarrow \mathcal{S}_{\mathsf{tid}} \setminus \{(\mathsf{pk}, \cdot, \cdot)\}$
  else $\mathsf{ip}, \mathsf{port} \leftarrow$ extract from request;
    $\mathcal{S}_{\mathsf{tid}} \leftarrow \mathcal{S}_{\mathsf{tid}} \cup \{(\mathsf{pk}, \mathsf{ip}, \mathsf{port})\}$
  $\mathcal{P} \leftarrow\!\!\$\; \mathcal{S}_{\mathsf{tid}} \setminus \{(\mathsf{pk}, \cdot, \cdot)\}$
  return $\mathcal{P}$
```

Figure 4. Torrent announcement with reputation-based access control. The tracker verifies the signature, computes reputation via Rep from on-chain statistics, enforces MinRep for new downloads, updates internal swarm state $\mathcal{S}_{\mathsf{tid}}$, and returns a random sample of peers.

pieces. Torrent metadata $\mathcal{T} = (h_\mathcal{T}, [h_1, h_2, \ldots, h_n])$ contains the infohash $h_\mathcal{T}$ and piece hashes $[h_1, \ldots, h_n]$. When peer A uploads piece p_i to B, peer B verifies piece integrity ($\mathsf{Hash}(p_i) = h_i$) then generates a cryptographic receipt via Attest (Fig. 5), signing a message that binds the infohash $h_\mathcal{T}$, sender's public key pk_A, piece hash h_i, piece index i, and epoch timestamp. Peer B returns this signed receipt to A. Uploaders collect receipts from downloaders as proof of contributions. Later, uploaders report accumulated receipts to the tracker via Report. The tracker verifies the aggregate signature from all downloaders, then credits the uploader's upload counter by the total piece size and each downloader's download counter. The contract serves as PKI: peers retrieve each other's public keys from on-chain registration records to verify receipt signatures.

Piece transfer attestation

```
Attest($\mathsf{sk}_{\mathsf{receiver}}, \mathsf{pk}_{\mathsf{sender}}, p_i, \mathcal{T}, t_{\textsc{epoch}}$):
  Parse $\mathcal{T} = (h_\mathcal{T}, [h_1, \ldots, h_n])$
  if $i \notin [1, n]$ then return $\perp$
  if $\mathsf{Hash}(p_i) \neq h_i$ then return $\perp$
  $m \leftarrow (h_\mathcal{T} \,\|\, \mathsf{pk}_{\mathsf{sender}} \,\|\, h_i \,\|\, i \,\|\, t_{\mathrm{epoch}})$
  $\sigma_{\mathsf{receipt}} \leftarrow \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathsf{receiver}}, m)$
  return $\sigma_{\mathsf{receipt}}$

Verify($\mathsf{pk}_{\mathsf{receiver}}, \mathsf{pk}_{\mathsf{sender}}, p_i, \mathcal{T}, t_{\textsc{epoch}}, \sigma_{\mathsf{receipt}}$):
  Parse $\mathcal{T} = (h_\mathcal{T}, [h_1, \ldots, h_n])$
  if $i \notin [1, n]$ then return 0
  if $\mathsf{Hash}(p_i) \neq h_i$ then return 0
  $m \leftarrow (h_\mathcal{T} \,\|\, \mathsf{pk}_{\mathsf{sender}} \,\|\, h_i \,\|\, i \,\|\, t_{\mathrm{epoch}})$
  return $\Sigma.\mathsf{Verify}(\mathsf{pk}_{\mathsf{receiver}}, m, \sigma_{\mathsf{receipt}})$
```

Figure 5. Piece transfer attestation with epoch-based double-spend resistance. Attest verifies piece integrity and generates a cryptographic receipt binding torrent infohash, sender public key, piece hash, piece index, and epoch. Verify checks signature and piece integrity. Epoch timestamps prevent receipt reuse.

Statistics reporting. Clients periodically report upload and download deltas via Report (Fig. 6). The report message m contains public keys from peers who acknowledged transfers, torrent metadata, and an aggregated signature combining all receipts. The tracker reconstructs signed messages and verifies the aggregate signature using Σ.AggVer, then retrieves current reputation values from the smart contract, adds the reported deltas, and writes the updated reputation back with authenticated updates.

Statistics reporting (in TEE)

```
Report($\mathsf{uid}, \mathsf{pk}, \{\mathsf{pk}_j\}_{j \in \mathcal{J}}, \mathcal{T}, \{t_j\}_{j \in \mathcal{J}}, \sigma_{\mathsf{a}}, \Delta_{\mathsf{up}}, \Delta_{\mathsf{down}}$):
  Parse $\mathcal{T} = (h_\mathcal{T}, [h_1, \ldots, h_n])$
  $t_{\mathrm{now}} \leftarrow$ current epoch
  for each $j \in \mathcal{J}$:
    $t_{\mathrm{epoch},j} \leftarrow \lfloor t_j / W \rfloor$
    if $t_{\mathrm{epoch},j} \notin [t_{\mathrm{now}} - \Delta, t_{\mathrm{now}}]$ then return 0
    $\mathsf{rid}_j \leftarrow (h_\mathcal{T}, \mathsf{pk}, \mathsf{pk}_j, h_j, j, t_{\mathrm{epoch},j})$
    if $\mathsf{rid}_j \in \mathcal{R}_{\mathrm{recent}}$ then return 0
    $m_j \leftarrow (h_\mathcal{T} \,\|\, \mathsf{pk} \,\|\, h_j \,\|\, j \,\|\, t_{\mathrm{epoch},j})$
  if $\Sigma.\mathsf{AggVer}(\{(\mathsf{pk}_j, m_j)\}_{j \in \mathcal{J}}, \sigma_{\mathsf{a}}) = 0$ then return 0
  $(\mathsf{up}, \mathsf{down}) \leftarrow \mathsf{SC.Read}(\mathsf{addr}, \mathsf{uid})$
  $\mathsf{up}' \leftarrow \mathsf{up} + \Delta_{\mathsf{up}}$; $\mathsf{down}' \leftarrow \mathsf{down} + \Delta_{\mathsf{down}}$
  $\mathsf{SC.Write}(\mathsf{addr}, \mathsf{uid}, (\mathsf{up}', \mathsf{down}'), \mathsf{auth}_{\mathsf{TEE}})$
  for each $j \in \mathcal{J}$:
    $\mathcal{R}_{\mathrm{recent}} \leftarrow \mathcal{R}_{\mathrm{recent}} \cup \{\mathsf{rid}_j\}$
    Retrieve downloader's user ID $\mathsf{uid}_j$ for $\mathsf{pk}_j$
    $(\mathsf{up}_j, \mathsf{down}_j) \leftarrow \mathsf{SC.Read}(\mathsf{addr}, \mathsf{uid}_j)$
    $\mathsf{down}'_j \leftarrow \mathsf{down}_j + \mathsf{piece\_size}$
    $\mathsf{SC.Write}(\mathsf{addr}, \mathsf{uid}_j, (\mathsf{up}_j, \mathsf{down}'_j), \mathsf{auth}_{\mathsf{TEE}})$
  return 1
```

Figure 6. Statistics reporting with epoch-based double-spend resistance. Receipts are checked for expiry and deduplication before batch verification via AggVer. IDs are stored in $\mathcal{R}_{\mathrm{recent}}$ with periodic garbage collection.

Reputation migration. When a tracker becomes unavailable, reputation migrates to a new instance via Migrate (Fig. 7). The new tracker generates a fresh instance ID and provides cryptographic attestation proving authenticity. After verification, migration creates a new smart contract referencing the old contract as predecessor, establishing a verifiable chain of custody. Reputation data remains immutable in the old contract and becomes accessible through the new contract's referrer link. Single-level migration prevents complex multi-hop inheritance while enabling tracker continuity.

Reputation migration (in TEE)

```
Migrate($\mathsf{addr}_{\mathsf{old}}, \pi$):
  Parse $\pi = (\mathsf{iid}_{\mathsf{new}}, \mathsf{auth}_{\mathsf{TEE}})$
  Verify TEE attestation $\mathsf{auth}_{\mathsf{TEE}}$ for new tracker instance
  if attestation is invalid then return $\perp$
  $\mathsf{params} \leftarrow (\mathsf{iid}_{\mathsf{new}}, \mathsf{addr}_{\mathsf{old}}, \mathsf{pk}_{\mathsf{tracker}}, \mathsf{auth}_{\mathsf{TEE}})$
  $\mathsf{addr}_{\mathsf{new}} \leftarrow \mathsf{SC.Init}(\mathsf{params})$
  return $\mathsf{addr}_{\mathsf{new}}$
```

Figure 7. Tracker migration with TEE attestation verification. Deploys a new smart contract referencing the old contract as predecessor, establishing single-hop inheritance that preserves reputation continuity.

Reputation Contract Layer. The *RepFactory* smart-contract architecture uses a factory pattern to provide data persistence, accountability, and confidentiality for user reputations across tracker instances. First, *persistence* ensures reputation persists despite tracker shutdowns or censorship. Committing all updates on-chain inherits the blockchain's durability and immutability. Second, *transferability* means that when a tracker migrates, reputation data remains portable. Each RepFactory-deployed contract follows a standardized interface, allowing a new contract to reference its predecessor as a *referrer*. Single-hop inheritance preserves user scores while preventing conflicts in multi-level merging. Third, *accountability* ensures only authenticated tracker instances can issue state transitions. The factory grants exclusive write privileges to the authenticated identity that deployed the contract. State updates become cryptographically attributable to verified tracker execution. Fourth, *confidentiality* maintains user identities as pseudonymous through opaque on-chain storage. The public reputation table only stores user id and public keys, which reveals no actual information about the underlying user. This design breaks the link between on-chain data and real-world identities, preventing deanonymizing users or tracking them across multiple tracker instances.

On-chain persistence with authenticated updates provides tamper resistance, accountability, and state continuity under migration. Reputation is auditable yet unlinkable.

Privacy and federation. Users can prove tracker membership by referencing the contract and providing either a signature or zero-knowledge membership proof. Reputation attributes (e.g., pass the reputation threshold) can be proven without revealing exact values or identity. Fresh public keys per contract prevent cross-tracker linkability. Federated tracker instances inherit from existing ones, for reputation continuity while maintaining pseudonymity.

4.3. Tracker-less peer discovery and file exchange

Private trackers typically disable DHT-based peer discovery because standard DHT protocols lack authentication and access control. Any peer can freely join a swarm, undermining reputation-based admission and access control enforcement. Consequently, private communities mark torrents as *private*, forcing all peer discovery to occur exclusively through a trusted tracker.

We extend Kademlia [28] with authentication to provide a DHT fallback preserving access control when the tracker is unavailable. The smart contract serves as PKI: registered users have on-chain public keys and reputation records. Peers authenticate DHT announcements using these credentials, admitting only users with sufficient reputation.

Kademlia is a distributed hash table that maps keys to values without centralized coordination. Each node has a 160-bit identifier, and data are stored on the nodes whose identifiers are closest to a given key under XOR distance $d(x, y) = x \oplus y$. To join, a peer contacts bootstrap nodes from list $\mathcal{B}$ to learn about other participants and builds a routing table of known nodes. To announce availability for torrent $h_{\mathcal{T}}$, peer P_i locates the k nodes closest to $h_{\mathcal{T}}$ (typically $k = 20$) through iterative lookups and stores its contact information $(\mathsf{pk}_i, \mathsf{ip}_i, \mathsf{port}_i)$ on those nodes. To discover peers, a client performs the same lookup and retrieves peer records from the closest nodes. These operations complete in $O(\log n)$ steps for n participants.

Each `.torrent` file includes bootstrap information $(\mathcal{B}, \mathsf{addr}_{\mathsf{rep}})$, where $\mathcal{B} = \{(\mathsf{ip}_i, \mathsf{port}_i)\}_{i \in [k]}$ lists DHT bootstrap nodes and $\mathsf{addr}_{\mathsf{rep}}$ specifies the reputation contract address. When the tracker becomes unavailable, peer P_i joins with node identifier $\mathsf{nodeID}_i = \mathsf{Hash}(\mathsf{pk}_i)$, binding each DHT identity to a verifiable on-chain user.

When peer P_i announces for torrent $h_{\mathcal{T}}$, it sends message $m = (\mathtt{announce} \parallel h_{\mathcal{T}} \parallel \mathsf{pk}_i \parallel \mathsf{ip}_i \parallel \mathsf{port}_i)$ with signature $\sigma_i = \mathsf{Sign}(\mathsf{sk}_i, m)$ to the k closest nodes. Each node n verifies the signature and queries the smart contract: $(\mathsf{pk}'_i, u_i, d_i) \leftarrow \mathsf{SC.Read}(\mathsf{addr}_{\mathsf{rep}}, \mathsf{uid}_i)$. Node n accepts the announcement only if $\mathsf{pk}'_i = \mathsf{pk}_i$ and $\mathsf{Rep}(u_i, d_i) \geq \mathsf{MinRep}$, then adds $(\mathsf{pk}_i, \mathsf{ip}_i, \mathsf{port}_i)$ to its peer list for $h_{\mathcal{T}}$.

Each peer P_i maintains a local view $\mathcal{S}^{(i)}_{\mathsf{local}} \subseteq \mathcal{S}$ of active peers for each torrent. When P_j announces to P_i, the protocol mirrors the centralized Announce procedure (Fig. 4), except that P_i uses its local view instead of a global tracker database. P_i verifies P_j's signature $\sigma_j = \mathsf{Sign}(\mathsf{sk}_j, \mathtt{announce} \parallel \mathsf{uid}_j \parallel h_{\mathcal{T}} \parallel \mathsf{event})$, checks on-chain registration and reputation, updates $\mathcal{S}^{(i)}_{\mathsf{local}}$, and returns a random sample $\mathcal{P} \leftarrow\!\!\$\ \mathcal{S}^{(i)}_{\mathsf{local}}$. Local views converge over time through authenticated DHT announcements, peer exchange (PEX), and periodic re-announcements. Peers can cache contact information from tracker responses during normal operation to build their own bootstrap node sets $\mathcal{B}_{\mathsf{cached}}$, ensuring rapid DHT network joining when the tracker becomes unavailable. By Kademlia's logarithmic routing properties, active peers remain discoverable in $O(\log n)$ hops.

File transfer follows the attestation protocol (see Section 4.2). Each piece transfer from P_s to P_r produces receipt $\sigma_{\mathsf{receipt}} = \mathsf{Attest}(\mathsf{sk}_r, \mathsf{pk}_s, p_i, \mathcal{T}, t_{\mathsf{epoch}})$. During tracker downtime, peers store receipts locally in $\mathcal{R} = \{(\mathsf{pk}_j, p_i, \mathcal{T}, \sigma_j)\}$. After recovery, peers submit accumulated receipts via Report to update on-chain reputation.

4.4. Optimized attestation

Per-piece BLS attestation ensures strong verifiability but introduces computational overhead during high-throughput transfers. We discuss several optimizations that reduce this signing overhead.

The frequency of cryptographic attestation can be adjusted based on trust dynamics during a transfer. At session start, trust between peers is not yet established: the sender has not demonstrated reliability and the receiver may defect. Frequent signatures during this phase provide strong accountability and enable early termination if either party misbehaves. As the session progresses and mutual trust builds through successful exchanges, signing frequency can decrease. Near session end, frequent signatures resume: the remaining pieces become highly valuable to the receiver, and the sender requires proof of delivery to claim full credit. A practical policy signs every piece during the first 100 transfers, every 10 pieces during the middle phase, and every piece for the final 100 transfers. For a transfer with n pieces (where $n > 200$), this requires $200 + \lceil (n - 200)/10 \rceil$ signatures instead of n, reducing overhead by approximately $(1 - \frac{200+(n-200)/10}{n}) \approx 0.9 - \frac{180}{n}$, approaching 90% reduction for large files while maintaining security at critical trust boundaries.

Established reputable peers can negotiate reduced signing frequencies. The tracker provides reputation scores during the Announce phase. High-reputation receivers may propose signing every k pieces, where k scales with reputation. Senders accept this proposal only if receivers' on-chain reputation exceeds a threshold. If a receiver later defects, senders report fraud using the partial attestations,

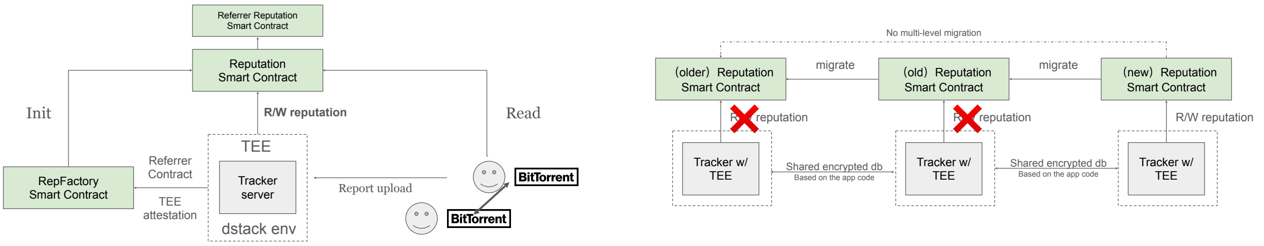

Figure 8. (a) Workflow of the reputation smart contract showing interactions among the tracker's TEE, the *RepFactory* contract, and BitTorrent clients for reputation creation, update, and read operations. (b) A persistence and migration scenario in which reputation data survives tracker failures through single-hop inheritance between smart contracts, ensuring long-term continuity of user reputations.

```
Smart Contract Functions

STATE VARIABLES:
    Owner: Address of the tracker, with the TEE used for authentication.
    Referrer: Address of the predecessor contract (optional), default setting to an empty address.
    Data: Mapping from $uid \to (up, down)$.

SC.Init(params): // RepFactory function
    Parse params $\to$ (iid, ref_addr, pk, auth).
    Verify auth matches the deployment key pk.
    Owner $\leftarrow$ (pk, auth).
    Referrer $\leftarrow$ ref_addr.
    Return new contract address $\mathsf{addr}_{\mathsf{new}}$.

SC.Read(uid):
    $(u, d) \leftarrow$ Data[uid].
    if $(u, d) = \perp \wedge$ Referrer $\neq \emptyset$ then:
        $(u, d) \leftarrow$ SC.Read(Referrer, uid) // Single-hop lookup
    else return $\perp$
    Return $(u, d)$.

SC.Write(uid, value, auth):
    if auth $\neq$ $\mathsf{Owner}_{\mathsf{auth}}$ then return $\perp$ // Access Control
    Data[uid] $\leftarrow$ value.
    Return success.
```

Figure 9. Function details of the Reputation Smart Contract implementing the abstract interface. It enforces access control via TEE authentication and handles data inheritance via the referrer.

and the tracker penalizes the receiver's reputation. This approach amortizes overhead for trusted peers while maintaining accountability through reputation at risk.

Rather than signing individual pieces, receivers can sign commitments to batches of pieces. A receiver computes a Merkle tree over piece hashes and signs the root after transferring k pieces. The sender verifies each piece against the torrent metadata during transfer and accepts the batch signature as proof of all k pieces. For $k = 10$, this reduces signature operations by 10×. The trade-off is reduced granularity: if the receiver defects mid-batch, the sender loses credit for transferred pieces. Batch size should be chosen based on piece value: smaller batches for high-value content, larger batches for bulk transfers.

For scenarios requiring per-piece attestation with minimal overhead, elliptic curve signatures provide an alternative to BLS. We employ ECDSA (secp256k1) [22], which offers significantly faster signing and verification than BLS while maintaining strong security guarantees. Each transfer session uses an ephemeral ECDSA keypair authenticated by the receiver's long-term BLS key.

Let Σ_{BLS} be the long-term signature scheme and Σ_{ECDSA} be the ECDSA scheme. At session start, the receiver generates $(\mathsf{sk}^{\mathsf{ECDSA}}, \mathsf{pk}^{\mathsf{ECDSA}})$ and signs $\mathsf{pk}^{\mathsf{ECDSA}}$ along with session metadata using their BLS key:

$$\begin{aligned} \mathsf{sid} &\leftarrow\$ \{0,1\}^{256} \\ m_0 &\leftarrow (\mathsf{sid} \parallel h_{\mathcal{T}} \parallel \mathsf{pk}_{\mathsf{sender}} \parallel \mathsf{pk}^{\mathsf{ECDSA}}) \\ \sigma_0 &\leftarrow \Sigma_{\mathsf{BLS}}.\mathsf{Sign}(\mathsf{sk}^{\mathsf{BLS}}_{\mathsf{receiver}}, m_0) \end{aligned}$$

For each piece p_i, the receiver signs $(h_{\mathcal{T}}, \mathsf{pk}_{\mathsf{sender}}, h_i, i)$ using $\mathsf{sk}^{\mathsf{ECDSA}}$. The sender verifies each signature immediately, preserving tit-for-tat. All per-piece signatures $\{\sigma_i\}_{i \in [n]}$ are collected and reported to the tracker, which verifies each signature individually. The tracker then credits the sender $n \times \mathsf{piece\_size}$ bytes.

When reporting transfers with multiple peers, the sender aggregates BLS session authentication signatures across peers. For sessions with peers indexed by $j \in \mathcal{J}$, each with session certificate $\mathsf{cert}_j = (\mathsf{sid}_j, h_{\mathcal{T}}, \mathsf{pk}_{\mathsf{sender}}, \mathsf{pk}_j^{\mathsf{ECDSA}})$ and ECDSA signatures $\{\sigma_{i,j}\}$ for transferred pieces, the sender computes:

$$\sigma_{\mathsf{agg}} \leftarrow \Sigma_{\mathsf{BLS}}.\mathsf{Agg}(\{(\mathsf{pk}_j, \mathsf{cert}_j, \sigma_{0,j})\}_{j \in \mathcal{J}})$$

The report contains one aggregated BLS signature plus all ECDSA per-piece signatures for each peer session.

Table 1 compares optimization approaches for a 5.1 GB torrent with 2,560 pieces (2 MB each).

Table 1. COMPARISON OF ATTESTATION OPTIMIZATIONS

Approach	Signatures	Sign Time[a]	Report Size[b]
Per-piece BLS	2,560	5.1s	2.3 MB[c]
Adaptive frequency	~512	1.02s	456 KB
Batch ($k = 10$)	256	0.51s	228 KB
ECDSA (per-piece)	2,560	0.53s	160 KB

[a] Sign time includes both generation and verification.
[b] Report size assumes BLS aggregation where applicable.
[c] After BLS aggregation: 32 bytes per piece hash plus one 96-byte agg. signature.

ECDSA provides fast per-piece attestation with moderate bandwidth overhead. With signing and verification approximately 10× faster than BLS (0.2ms per piece compared to 2ms), ECDSA reduces computational overhead by 90% while maintaining full per-piece verification during transfer. The report size of 160 KB for 2,560 pieces represents a 30× reduction compared to per-piece BLS, though larger than batched approaches. Adaptive and batch approaches trade granularity for further reduced computational and bandwidth overhead. The choice depends on trust model and performance requirements: ECDSA for strong per-piece accountability with low computational cost, adaptive frequency for established peer relationships, and batching for bulk transfers where occasional disputes are acceptable. For real-time streaming applications requiring full verification, ECDSA provides the best balance of security and performance.

5. Security analysis

We now formally analyze the security properties of our Persistent BitTorrent Tracker System (PBTS). Our security model addresses the core threats identified in Section 3: reputation manipulation, false reporting, and unauthorized tracker operations. Each property is defined through a security game between a challenger and a probabilistic polynomial-time (PPT) adversary $\mathcal{A}$, capturing the adversary's advantage through a probability expression.

This section relies on three assumptions: *(1)* The signature scheme Σ provides existential unforgeability under chosen message attacks (EUF-CMA) and supports secure signature aggregation. *(2)* The TEE provides correct execution with secure attestation, ensuring that tracker code runs as specified. *(3)* The smart contract layer ensures integrity of state transitions, rejecting unauthorized writes.

We analyze four security properties. *Registration authenticity* (Section 5.1) prevents identity impersonation by requiring valid signatures from key holders during account creation. *Receipt non-repudiation* (Section 5.2) makes it impossible for peers to deny having received data after signing acknowledgments. *Report soundness* (Section 5.3) bounds reputation inflation: users can only claim credit supported by valid receipts from other peers. *Receipt non-reusability* (Section 5.4) prevents double-spending attacks where the same receipt appears in multiple reports.

These properties guarantee that reputation scores reflect actual contributions, the on-chain state remains consistent with real file transfers, and malicious users gain no advantage over honest participants.

5.1. Registration authenticity

The first requirement for a secure reputation system is that identities cannot be forged or stolen. In PBTS, each user registers by proving ownership of a public key through a digital signature. This prevents adversaries from registering under someone else's public key or creating accounts without corresponding secret keys, which would enable various attacks such as reputation theft or Sybil identity creation without accountability.

Definition 5.1 (Registration authenticity)**.** *A PBTS scheme satisfies registration authenticity if for any PPT adversary $\mathcal{A}$, the probability*

$$\Pr\left[\begin{array}{l}\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda);\\ (\mathsf{uid}^*, \mathsf{pk}^*, \sigma^*, \mathsf{params}^*) \leftarrow \mathcal{A}^{\mathsf{Register}(\cdot)}(\mathsf{pp}) :\\ \mathsf{Register}(\mathsf{uid}^*, \mathsf{pk}^*, \sigma^*, \mathsf{params}^*) = 1\\ \wedge\, \mathsf{pk}^* \notin \mathcal{Q}_{\mathsf{reg}}\end{array}\right]$$

is negligible in λ, where $\mathcal{Q}_{\mathsf{reg}}$ is the set of public keys that $\mathcal{A}$ submitted to Register *queries (i.e., keys for which the adversary generated or obtained the corresponding secret keys).*

Theorem 5.2. *If Σ is an EUF-CMA secure signature scheme and the TEE provides correct execution and attestation, then the PBTS construction from Section 4.2 satisfies registration authenticity.*

Proof. We prove by reduction to the EUF-CMA security of Σ. Suppose there exists a PPT adversary $\mathcal{A}$ that breaks registration authenticity with non-negligible advantage ϵ. We construct a PPT algorithm $\mathcal{B}$ that uses $\mathcal{A}$ to break the EUF-CMA security of Σ with advantage ϵ.

$\mathcal{B}$ receives a challenge public key pk^* from the EUF-CMA challenger and has access to a signing oracle $\mathcal{O}_{\mathsf{Sign}}(\mathsf{sk}^*, \cdot)$. $\mathcal{B}$ runs $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$ and gives pp to $\mathcal{A}$.

When $\mathcal{A}$ makes registration queries, $\mathcal{B}$ responds like so:

- For queries with $\mathsf{pk} \neq \mathsf{pk}^*$: $\mathcal{B}$ generates fresh key pairs $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}()$ and processes registration normally, recording pk in $\mathcal{Q}_{\mathsf{reg}}$.
- For queries involving pk^*: $\mathcal{B}$ uses its signing oracle $\mathcal{O}_{\mathsf{Sign}}$ to generate the registration signature, but does not add pk^* to $\mathcal{Q}_{\mathsf{reg}}$ since $\mathcal{B}$ does not know sk^*.

When $\mathcal{A}$ outputs $(\mathsf{uid}^*, \mathsf{pk}^*, \sigma^*, \mathsf{params}^*)$ with $\mathsf{pk}^* \notin \mathcal{Q}_{\mathsf{reg}}$ and $\mathsf{Register}(\mathsf{uid}^*, \mathsf{pk}^*, \sigma^*, \mathsf{params}^*) = 1$, the registration algorithm (Fig. 3) verifies:

$$\begin{aligned} m^* &= (\texttt{register} \parallel \mathsf{iid} \parallel \mathsf{uid}^*)\\ \Sigma.&\mathsf{Verify}(\mathsf{pk}^*, m^*, \sigma^*) = 1\end{aligned}$$

Since $\mathsf{pk}^* \notin \mathcal{Q}_{\mathsf{reg}}$ and m^* was not queried to $\mathcal{O}_{\mathsf{Sign}}$, the pair (m^*, σ^*) constitutes a valid signature forgery. $\mathcal{B}$ outputs this forgery, breaking the EUF-CMA security of Σ with advantage ϵ and contradicting the assumed EUF-CMA security of Σ, so ϵ must be negligible. $\square$

5.2. Receipt non-repudiation

A key component of our system is the p2p attestation mechanism, where receiving peers sign cryptographic receipts acknowledging piece transfers. For this to be meaningful, receipts must be non-repudiable: one cannot credibly deny receiving data after producing a valid receipt.

Definition 5.3 (Receipt non-repudiation)**.** *A PBTS scheme satisfies receipt non-repudiation if for any PPT adversary $\mathcal{A}$, the probability*

$$\Pr\left[\begin{array}{l} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{MinRep}, \mathsf{InitCredit}); \\ (\mathsf{sk}_S, \mathsf{pk}_S) \leftarrow \mathsf{KeyGen}(); \\ (p_i, \mathcal{T}, t_{epoch}, \mathsf{sk}_\mathcal{A}, \mathsf{pk}_\mathcal{A}) \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{pk}_S); \\ \sigma_{\mathsf{receipt}} \leftarrow \mathsf{Attest}(\mathsf{sk}_\mathcal{A}, \mathsf{pk}_S, p_i, \mathcal{T}, t_{epoch}); \\ \mathsf{b} \leftarrow \mathsf{Report}(\mathsf{uid}_S, \mathsf{pk}_S, \{\mathsf{pk}_\mathcal{A}\}, \mathcal{T}, \\ \qquad \{t_{epoch}\}, \sigma_{\mathsf{receipt}}, \Delta_{\mathsf{up}}, 0) : \\ \mathsf{Verify}(\mathsf{pk}_\mathcal{A}, \mathsf{pk}_S, p_i, \mathcal{T}, t_{epoch}, \\ \qquad \sigma_{\mathsf{receipt}}) = 1 \wedge \mathsf{b} = 0 \end{array}\right]$$

is negligible in λ. The adversary wins if Attest *produces a receipt $\sigma_{\mathsf{receipt}}$ that verifies but causes an honest sender's* Report *to fail (successful repudiation).*

Theorem 5.4 (Receipt non-repudiation)**.** *If Σ is EUF-CMA secure, the TEE executes* Report *correctly, and the smart contract enforces access control, then the PBTS construction satisfies receipt non-repudiation.*

Proof. Suppose adversary $\mathcal{A}$ wins the non-repudiation game with non-negligible advantage ϵ. Then $\mathcal{A}$ produces a receipt $\sigma_{\mathsf{receipt}} = \mathsf{Attest}(\mathsf{sk}_\mathcal{A}, \mathsf{pk}_S, p_i, \mathcal{T}, t_{\mathrm{epoch}})$ that satisfies $\mathsf{Verify}(\mathsf{pk}_\mathcal{A}, \mathsf{pk}_S, p_i, \mathcal{T}, t_{\mathrm{epoch}}, \sigma_{\mathsf{receipt}}) = 1$, but the honest sender's Report call returns 0.

The Report algorithm (Fig. 6) rejects a report only if signature verification fails, the timestamp t_{epoch} is outside the valid epoch window $[t_{\mathrm{now}} - \Delta, t_{\mathrm{now}}]$, the receipt has been used before (double-spending with rid $\in \mathcal{R}_{\mathrm{recent}}$), or the piece hash does not match ($h_i \neq \mathsf{Hash}(p_i)$).

By the winning condition, signature verification succeeds, so the first condition does not hold. For an honest sender immediately reporting a fresh transfer, t_{epoch} is current and within the valid window. The receipt is used for the first time, so rid $\notin \mathcal{R}_{\mathrm{recent}}$. The honest sender uses the actual p_i transferred by $\mathcal{A}$, so $h_i = \mathsf{Hash}(p_i)$ holds by construction.

Since none of the rejection conditions hold and the TEE executes Report correctly by assumption, the algorithm must return 1, contradicting the assumption that $\mathcal{A}$ wins with Report returning 0. Therefore, ϵ must be negligible. □

5.3. Report soundness

With registration authenticity and receipt non-repudiation established, we now address the core security property: users cannot inflate reputation beyond their actual contributions. This property prevents false reporting by requiring that every transfer be backed by a cryptographic acknowledgment from the counterparty.

Definition 5.5 (Report soundness)**.** *A PBTS scheme satisfies report soundness if for any PPT adversary $\mathcal{A}$ that interacts with honest peers and the tracker, the probability*

$$\Pr\left[\begin{array}{l} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda); \\ (\mathsf{uid}_\mathcal{A}, \mathsf{pk}_\mathcal{A}) \leftarrow \mathcal{A}^{\mathsf{Register}(\cdot), \mathsf{Announce}(\cdot), \mathit{Peers}}(\mathsf{pp}); \\ (\mathsf{up}_{true}, \mathsf{down}_{true}) \leftarrow \mathsf{TrueStats}(\mathcal{A}); \\ \mathsf{Report}(\mathsf{uid}_\mathcal{A}, \mathsf{pk}_\mathcal{A}, \{\mathsf{pk}_j\}, \mathcal{T}, \sigma_{\mathsf{agg}}, \Delta_{\mathsf{up}}, \Delta_{\mathsf{down}}) = 1; \\ (\mathsf{up}_{claimed}, \mathsf{down}_{claimed}) \leftarrow \mathsf{SC.Read}(\mathsf{addr}, \mathsf{uid}_\mathcal{A}) : \\ \mathsf{up}_{claimed} > \mathsf{up}_{true} \vee \mathsf{down}_{claimed} < \mathsf{down}_{true} \end{array}\right]$$

is negligible in λ, where $\mathsf{TrueStats}(\mathcal{A})$ *tracks actual data uploaded and downloaded by $\mathcal{A}$ to/from honest peers.*

Theorem 5.6. *If Σ satisfies aggregate unforgeability, the TEE executes the tracker code correctly, and the smart contract enforces access control, then the PBTS construction satisfies report soundness.*

Proof. The Report algorithm (Fig. 6) accepts a report only if the aggregated signature σ_{agg} verifies correctly over the set of receipts $\{(\mathsf{pk}_j, m_j)\}_{j \in \mathcal{J}}$, where each $m_j = (h_\mathcal{T} \parallel \mathsf{pk}_\mathcal{A} \parallel h_j \parallel j \parallel t_j)$ represents a receipt from peer j acknowledging receipt of piece j from $\mathcal{A}$ at time t_j.

We analyze two cases:

Case 1: Over-reporting uploads ($\mathsf{up}_{\mathbf{claimed}} > \mathsf{up}_{\mathbf{true}}$).

For $\mathcal{A}$ to claim credit exceeding its actual uploads, it must provide valid receipts for pieces it never sent. This requires forging receipts from honest peers who never signed them.

Let $\mathcal{B}$ be a PPT algorithm attacking aggregate unforgeability. $\mathcal{B}$ receives challenge public keys $\{\mathsf{pk}_1^*, \ldots, \mathsf{pk}_k^*\}$ for k honest peers and access to signing oracles $\{\mathcal{O}_{\mathsf{Sign}}(\mathsf{sk}_i^*, \cdot)\}_{i \in [k]}$. $\mathcal{B}$ simulates the PBTS environment for $\mathcal{A}$, using the challenge keys as honest peers' public keys and the signing oracles to generate legitimate receipts when $\mathcal{A}$ uploads to honest peers.

When $\mathcal{A}$ submits a report claiming $\mathsf{up}_{\mathrm{claimed}} > \mathsf{up}_{\mathrm{true}}$, the report includes an aggregate signature σ_{agg} over receipts $\{(\mathsf{pk}_j, m_j)\}_{j \in \mathcal{J}}$. Since $\mathcal{A}$ over-reported uploads, at least one (pk_j^*, m_j^*) must correspond to an upload that never occurred, meaning m_j^* was never queried to $\mathcal{O}_{\mathsf{Sign}}(\mathsf{sk}_j^*, \cdot)$. Thus σ_{agg} constitutes a forgery, and $\mathcal{B}$ outputs it to break aggregate unforgeability with the same advantage as $\mathcal{A}$.

Case 2: Under-reporting downloads ($\mathsf{down}_{\mathbf{claimed}} < \mathsf{down}_{\mathbf{true}}$).

When $\mathcal{A}$ downloads piece p_i from an honest peer P, it must generate $\sigma_{\mathsf{receipt}} = \mathsf{Attest}(\mathsf{sk}_\mathcal{A}, \mathsf{pk}_P, p_i, \mathcal{T}, t_{\mathrm{epoch}})$ and return it to P. If $\mathcal{A}$ refuses to do so, honest peers detect non-cooperation and stop serving $\mathcal{A}$ (tit-for-tat enforcement). If $\mathcal{A}$ provides receipts, those can be submitted by honest uploaders, accurately tracking $\mathcal{A}$'s downloads.

Combining both cases, by Σ's aggregate unforgeability:

$$\begin{aligned} \mathsf{Adv}_{\mathsf{PBTS},\mathcal{A}}^{\mathsf{report}}(\lambda) &\leq \mathsf{Adv}_{\Sigma,\mathcal{B}}^{\mathsf{agg-forge}}(\lambda) \\ &= \mathsf{negl}(\lambda) \end{aligned}$$

□

5.4. Receipt non-reusability

Even with report soundness ensuring that receipts correspond to genuine transfers, another threat remains: receipt reuse. An adversary might attempt to submit the same receipt multiple times across different reports, effectively claiming credit for the same upload repeatedly.

Definition 5.7 (Receipt non-reusability)**.** *A PBTS scheme satisfies receipt non-reusability if for any PPT adversary $\mathcal{A}$, the following probability is negligible in λ:*

$$\Pr\left[\begin{array}{l}\mathsf{pp} \leftarrow \mathsf{Setup}(1^{\lambda});\\ \sigma_{\mathsf{receipt}} \leftarrow \mathcal{A}^{\mathsf{Register}(\cdot),\mathsf{Announce}(\cdot),\mathsf{Report}(\cdot)}(\mathsf{pp});\\ (R_1, R_2) \leftarrow \mathcal{A}(\sigma_{\mathsf{receipt}}):\\ \sigma_{\mathsf{receipt}} \in R_1 \wedge \sigma_{\mathsf{receipt}} \in R_2\\ \wedge\, \mathsf{Report}(R_1) = 1 \wedge \mathsf{Report}(R_2) = 1\end{array}\right]$$

Theorem 5.8. *If Σ is EUF-CMA secure and receipts include timestamps, then the PBTS construction satisfies receipt non-reusability.*

Proof. Each receipt includes an epoch identifier in the signed message, that is, if t_{epoch} represents the period during which the transfer occurred, we have:

$$m = (h_{\mathcal{T}} \parallel \mathsf{pk}_{\mathsf{sender}} \parallel h_i \parallel i \parallel t_{\mathsf{epoch}})$$

Time is divided into discrete epochs (e.g., hour-long windows), and $t_{\mathsf{epoch}} = \lfloor t_{\mathsf{current}}/W \rfloor$ where W is the epoch width and t_{current} is the time when the receipt is generated. The epoch timestamp is fixed at receipt generation: honest receivers sign the current epoch, so t_{epoch} cannot be greater than the time t_1 when the receipt first appears in a report. The epoch timestamp is provided as input to the Attest algorithm and signed by the receiver.

The tracker enforces two mechanisms: (1) temporal validity, accepting only receipts with recent timestamps, and (2) short-term deduplication via a set $\mathcal{R}_{\mathsf{recent}}$ of recently used receipt identifiers. Each receipt is uniquely identified by $(h_{\mathcal{T}}, \mathsf{pk}_{\mathsf{sender}}, \mathsf{pk}_{\mathsf{receiver}}, h_i, i, t_{\mathsf{epoch}})$.

Say adversary $\mathcal{A}$ successfully reuses receipt $\sigma_{\mathsf{receipt}}$ with time t_{epoch} by getting it accepted in reports R_1 and R_2 processed at times t_1 and t_2 where $t_1 < t_2$. Let the acceptance window at time t cover epochs in range $[t-\Delta, t]$.

Case 1: $t_2 \leq t_1 + \Delta + 1$**.** The reports are within $\Delta + 1$ epochs of each other. After R_1 is processed at epoch t_1, the receipt identifier is added to $\mathcal{R}_{\mathsf{recent}}$. Since $t_2 \leq t_1 + \Delta + 1$, the receipt remains in $\mathcal{R}_{\mathsf{recent}}$ when R_2 is processed. The deduplication check detects the reuse and rejects R_2.

Case 2: $t_2 > t_1 + \Delta + 1$**.** For the receipt to be valid at time t_2, we need $t_{\mathsf{epoch}} \geq t_2 - \Delta$. However, since the receipt was valid at time t_1, we have $t_{\mathsf{epoch}} \leq t_1 < t_2 - \Delta - 1 < t_2 - \Delta$. This contradicts the requirement for acceptance at t_2, so the receipt is rejected as expired.

The adversary cannot forge receipts with future timestamps because honest receivers sign the current timestamp at generation time. Forging such a signature contradicts the EUF-CMA security of Σ. Similarly, modifying the timestamp in an existing receipt invalidates the signature.

Since both cases lead to rejection:

$$\mathsf{Adv}^{\mathsf{reuse}}_{\mathsf{PBTS},\mathcal{A}}(\lambda) = 0$$

$\square$

6. Implementation

While the preceding sections focus on the protocol design and formal properties, practical deployment of the Persistent BitTorrent Tracker System (PBTS) requires careful attention to implementation challenges. In this section, we highlight general strategies for realizing PBTS in production, with particular attention to tracker liveness and duplication, secure interaction with the blockchain, and operational best practices.

6.1. Tracker liveness and duplication

A key challenge in TEE-based systems is maintaining availability in the presence of node failures, upgrades, or targeted censorship. Traditional TEE deployments derive cryptographic keys from hardware-bound sources, making state recovery and failover non-trivial. To address this, PBTS decouples key generation from physical hardware by utilizing an external KMS.

In our implementation, the KMS operates as a decentralized, attested service. Each tracker instance obtains its unique *Tracker Root Key* (key_{trk}) from KMS, where the key is deterministically derived from the cryptographic measurement of the tracker program and its configuration. This proceeds as follows:

(1) The tracker instance presents its TEE attestation report to the KMS, which verifies the measurement against an allowlist of approved binaries.

(2) Upon successful attestation, the KMS derives and returns key_{trk} for this tracker code/configuration.

(3) All subsequent ephemeral keys (for signing smart contract updates, encrypting tracker state, or authenticating with peers) are derived from key_{trk} using standardized key derivation functions.

This approach yields several operational benefits. First, *stateless recovery* enables seamless failover: if a tracker instance fails or is taken down, a new instance can be launched elsewhere, present the same attestation to the KMS, and retrieve key_{trk}, thus resuming operation with full access to persistent state and on-chain credentials. Second, *liveness via duplication* allows multiple tracker instances, each running the identical TEE-measured binary, to operate in parallel (e.g., for load balancing or geo-redundancy). Because they share key_{trk}, their operations remain consistent and interchangeable. Third, *censorship resistance* ensures the tracker is resilient against targeted takedowns, as state and control are not tied to any single machine or physical location.

6.2. Secure blockchain RPC and state integrity

For PBTS to provide accountability and persistence guarantees, tracker instances must maintain reliable and secure interactions with the blockchain hosting the reputation smart contracts. This requirement is twofold: ensuring *liveness* (the tracker can always read/write to the smart contract) and *integrity* (the tracker verifies that data read from the blockchain is authentic and untampered).

Our implementation adopts the following strategies. First, *RPC endpoint redundancy* configures the tracker with a dynamic list of blockchain RPC endpoints, potentially spanning multiple providers and geographies. If an endpoint becomes unavailable or censored, the tracker falls over to alternatives. Second, *light client verification* ensures that rather than relying solely on unverified RPC responses, the tracker runs a lightweight blockchain client (e.g., an SPV client) within the TEE. This client verifies block headers and Merkle proofs for all contract state reads, ensuring that returned values are consistent with canonical chain state and not subject to equivocation or censorship by RPC providers. Third, *authenticated writes* sign all contract write operations (e.g., updating reputation) using tracker-specific keys derived from key_{trk}. Smart contracts enforce access control, ensuring that only legitimate, attested tracker instances can mutate state. Fourth, *resilience to chain reorganizations* monitors chain finality and handles reorgs gracefully, ensuring that all on-chain state transitions are consistent and auditable.

This architecture ensures that blockchain interactions are robust against endpoint failures, adversarial RPC providers, and network-level censorship, while maintaining integrity and auditability of on-chain reputation data.

7. Evaluation

We evaluate the performance and practicality of our Persistent BitTorrent Tracker System (PBTS) through a series of micro-benchmarks and system-level simulations. Our evaluation aims to answer three key questions: *(1)* What is the computational overhead of the cryptographic primitives, particularly the receipt generation and verification? *(2)* How does the TEE environment impact the performance of tracker operations? *(3)* What is the end-to-end impact on client download performance, and how effectively does batching mitigate this overhead?

7.1. Experiment Setup

All experiments were conducted on a confidential virtual machine hosted on Phala Cloud [31], equipped with 2 vCPUs and 4 GB of RAM. The environment supports Intel TDX for TEE execution. We implemented the core PBTS components in Python, using the BLS12-381 curve [5]. We selected BLS specifically for its support of efficient signature aggregation and verification [17], which serves as the foundation for our batching optimizations. Our implementation is open-sourced and available online.[2]

To ensure backward compatibility with existing BitTorrent clients and trackers, we implemented receipt generation and verification as BEP10 extensions.

We focus our evaluation on the cryptographic and network overheads of the tracker and client. We do not benchmark smart contract operations (e.g., reputation persistence) as they are inherent to the underlying blockchain infrastructure and can be conducted asynchronously. The tracker commits state updates to the blockchain in the background, ensuring that block confirmation times do not block real-time peer interactions.

7.2. Micro-benchmarks

We first measure the baseline costs of the cryptographic operations that form the building blocks of our system. Table 2 summarizes the latency of key operations.

Table 2. MICRO-BENCHMARK RESULTS (MEAN LATENCY).

Operation	**Time (ms)**
Cryptographic Primitives	
BLS Keypair Generation	7.54
Receipt Creation (Sign)	134.19
Receipt Verification	340.92
TEE Operations	
Regular Key Gen (No TEE)	6.06
TEE Key Gen	18.01
Attestation Generation	17.34
Attestation Verification	1030.16

Receipt operations. Creating a cryptographic receipt involves a BLS signature operation, which takes approximately 134 ms. Verification is more expensive, requiring about 341 ms per receipt. These costs are non-trivial, highlighting the importance of our optimization strategies (batching and aggregation) discussed in Section 4.4.

TEE Overhead. Running code inside the TEE introduces measurable overhead. Key generation inside the TDX enclave takes roughly 18 ms, compared to 6 ms in a standard environment: a 3× increase. However, this absolute cost remains low enough not be a bottleneck for user registration. Attestation generation is efficient ($\approx$ 17 ms), while verification is computationally intensive ($\approx$ 1 s), but this is a one-time cost paid during tracker registration or migration, not per-transaction.

7.3. Client Download Performance

To understand the impact on user experience, we simulated file downloads under various network conditions and piece sizes. The primary metric is *throughput reduction*: the percentage loss in effective download speed due to the time spent generating and exchanging receipts.

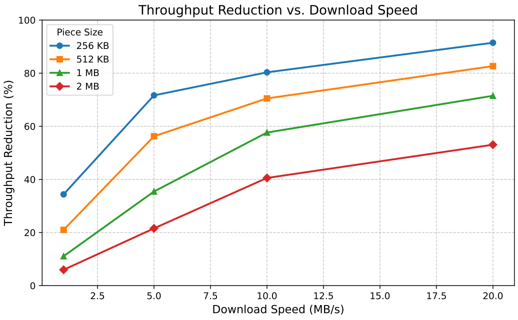

Figure 10. Throughput reduction as a function of download speed under different piece sizes (Batch Size = 1). Smaller pieces incur higher overhead due to more frequent signing operations.

Concrete example. Consider a user downloading a 1 GB file at 1 MB/s. In a standard BitTorrent system, this download would take approximately 1000 seconds. With PBTS enabled and using a standard piece size of 256 KB,

---

[2] https://anonymous.4open.science/r/pbts-75AC/

the client must generate and sign 4096 receipts. The computational overhead increases the download time by 52%, to approximately 1524 seconds. However, by increasing the piece size to 2 MB, the number of receipts drops to 512. The total download time becomes approximately 1063 seconds, representing only a 6% overhead. This demonstrates that with appropriate configuration (larger pieces), the impact on user experience is minimal.

Impact of network speed. At higher download speeds (e.g., 20 MB/s), the computational cost of receipt generation becomes a bottleneck. With 256 KB pieces, the system struggles to keep up, resulting in over 90% throughput loss. This necessitates the use of batching or probabilistic verification for high-bandwidth peers.

7.4. Optimization via batching

We evaluate signature aggregation and batching efficacy. Receipt aggregation allows amortizing verification cost.

Table 3. EFFECT OF BATCHING ON VERIFICATION SPEED.

Batch Size	Agg. Verify (ms)	Speedup
10	1293.18	2.55×
25	3959.54	2.07×
50	6535.18	2.23×
100	14230.40	2.07×

Table 3 shows that aggregating signatures provides a consistent speedup of roughly 2× to 2.5× compared to verifying them individually. While the absolute time for aggregate verification increases with batch size, the per-receipt cost drops significantly. For example, verifying a batch of 50 receipts takes ≈ 6.5 s, whereas verifying them individually would take ≈ 17 s (50×341 ms).

7.5. Analytical evaluation of optimizations

While our prototype implementation focuses on the core PBTS protocol, we analytically evaluate the impact of the optimizations proposed in Section 4.4 using the micro-benchmark data from Table 2. We project the computational overhead for three optimization strategies: *adaptive frequency* reduces signing operations by $(1 - \frac{200+(n-200)/10}{n})$ for files with n pieces (signing every piece initially and finally, but every 10th piece in between), approaching 90% reduction for large files; *batch signing* signs a Merkle root for every 10 pieces ($k = 10$); and *ECDSA* replaces BLS with faster elliptic curve signatures for per-piece attestation.

Figure 11 illustrates the projected reduction in computational overhead. The baseline represents the cost of signing and verifying every piece using BLS. *Adaptive frequency* reduces the overhead to approximately 10% of the baseline for large files by skipping operations during the middle phase of the transfer, suitable for sessions where trust builds over time. *Batching ($k = 10$)* achieves a 10× reduction in signing costs and amortizes verification, resulting in $\approx$ 10% of the baseline overhead. *ECDSA* provides the most dramatic improvement for per-piece attestation. With signing and verification operations $\approx 10\times$ faster than BLS, it reduces the total overhead to approximately 10% of the baseline while maintaining the security property of per-piece attestations.

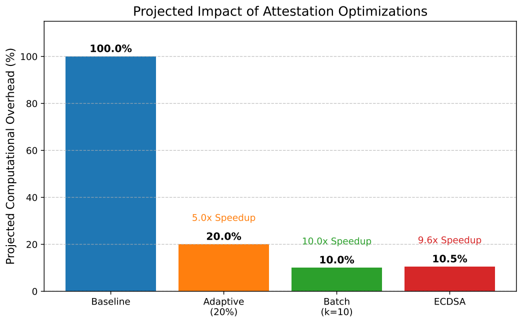

Figure 11. Projected computational overhead of proposed optimizations relative to baseline (per-piece BLS signing). ECDSA and batching offer significant performance improvements, reducing overhead by over 90%.

These projections indicate that the optimizations can effectively eliminate the computational bottleneck identified in Section 7.3, making PBTS viable even for high-bandwidth connections (e.g., >100 MB/s) where the baseline implementation would incur prohibitive overhead.

7.6. Discussion

Our evaluation demonstrates that while we introduce cryptographic mechanisms, the system remains practical for real-world deployment. The TEE overhead is negligible for infrequent operations like registration. For data transfer, using larger piece sizes (e.g., 2 MB) or batching receipts keeps the throughput penalty within acceptable limits (<10% overhead for typical broadband speeds). The security benefits, censorship resistance, and verifiability are achieved with a manageable performance trade-off.

8. Conclusion

This paper presents the Persistent BitTorrent Tracker System (PBTS), addressing the 3 critical weaknesses of private trackers: non-portable reputation, centralized points of failure, and unverifiable self-reports, This is done through smart contracts, cryptographic receipts, and TEE-based execution with authenticated DHT fallback. We formalize the scheme, prove four security properties, and demonstrate less than 6% throughput overhead on Intel TDX with 2.5× verification speedup via signature aggregation. Future work should explore privacy-preserving reputation proofs using zero-knowledge techniques, cross-chain reputation portability, and formal analysis of economic incentives under collusion. Beyond BitTorrent, PBTS demonstrates a general pattern for upgrading centralized coordination systems to Web3 architectures: blockchain-based persistent storage combined with cryptographic attestation and TEE-based execution provides censorship resistance with verifiable guarantees and acceptable performance trade-offs.

References

[1] Eytan Adar and Bernardo A. Huberman. “Free riding on Gnutella”. en. In: *First Monday* (Oct. 2000). ISSN: 1396-0466. DOI: 10.5210/fm.v5i10.792.

[2] AMD. *Secure Encrypted Virtualization API Version 0.24*. 2020. URL: https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/programmer-references/55766_SEV-KM_API_Specification.pdf.

[3] Cecylia Bocovich, Arlo Breault, David Fifield, Serene, and Xiaokang Wang. “Snowflake, a censorship circumvention system using temporary WebRTC proxies”. In: *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 2635–2652. ISBN: 978-1-939133-44-1. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/bocovich.

[4] Edward Bortnikov, Maxim Gurevich, Idit Keidar, Gabriel Kliot, and Alexander Shraer. “Brahms: byzantine resilient random membership sampling”. In: *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC 2008, Toronto, Canada, August 18-21, 2008*. Ed. by Rida A. Bazzi and Boaz Patt-Shamir. ACM, 2008, pp. 145–154. DOI: 10.1145/1400751.1400772.

[5] Sean Bowe. *New SNARK Curve*. https://electriccoin.co/blog/new-snark-curve/. Accessed: 2025-01-19. 2017.

[6] David Brooks and David Aslanian. *BitTorrent Protocol Abuses*. 2009. URL: https://www.blackhat.com/presentations/bh-usa-09/BROOKS/BHUSA09-Brooks-BitTorrHacks-PAPER.pdf.

[7] Vitalik Buterin. *Some ways to use ZK-SNARKs for privacy*. June 2022. URL: https://vitalik.eth.limo/general/2022/06/15/using_snarks.html.

[8] X. Chen, Y. Jiang, and X. Chu. “Measurements, Analysis and Modeling of Private Trackers”. In: *2010 IEEE Tenth International Conference on Peer-to-Peer Computing (P2P)*. 2010, pp. 1–10. DOI: 10.1109/P2P.2010.5569968.

[9] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. “Intel TDX Demystified: A Top-Down Approach”. In: *ACM Comput. Surv.* 56.9 (Apr. 2024), 238:1–238:33. ISSN: 0360-0300. DOI: 10.1145/3652597.

[10] Yukun Cheng, Xiaotie Deng, and Yuhao Li. “Study on Agent Incentives for Resource Sharing on P2P Networks”. In: *Asia-Pacific Journal of Operational Research* 39.03 (June 2022), p. 2150031. ISSN: 0217-5959. DOI: 10.1142/S0217595921500317.

[11] Yukun Cheng, Xiaotie Deng, Yuhao Li, and Xiang Yan. “Tight incentive analysis of Sybil attacks against the market equilibrium of resource exchange over general networks”. In: *Games and Economic Behavior* 148 (Nov. 2024), pp. 566–610. ISSN: 0899-8256. DOI: 10.1016/j.geb.2024.10.009.

[12] Yukun Cheng, Xiaotie Deng, Qi Qi, and Xiang Yan. “Truthfulness of a Network Resource-Sharing Protocol”. In: *Mathematics of Operations Research* 48.3 (Aug. 2023), pp. 1522–1552. ISSN: 0364-765X. DOI: 10.1287/moor.2022.1310.

[13] Bram Cohen. *Incentives Build Robustness in BitTorrent*. May 2003. URL: https://stuker.com/wp-content/uploads/import/i-1fd3ae7c5502dfddfe8b2c7acdefaa5e-bittorrentecon.pdf.

[14] Jon Dolan, Rob Levine, Ben Sisario, and Douglas Wolk. *The Powergeek 25 — the Most Influential People in Online Music - Blender*. 2007. URL: https://web.archive.org/web/20101221224758/http://www.blender.com/lists/68786/powergeek-25-151-most-influential-people-in-online-music.html?p=2.

[15] John R. Douceur. “The Sybil Attack”. In: *Peer-to-Peer Systems*. Ed. by Peter Druschel, Frans Kaashoek, and Antony Rowstron. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 251–260. ISBN: 978-3-540-45748-0.

[16] P. Druschel and A. Rowstron. “PAST: A Large-Scale, Persistent Peer-to-Peer Storage Utility”. In: *Proceedings Eighth Workshop on Hot Topics in Operating Systems*. May 2001, pp. 75–80. DOI: 10.1109/HOTOS.2001.990064.

[17] Benjamin Edgington. *BLS12-381 Aggregation*. https://hackmd.io/@benjaminion/bls12-381#Aggregation. Accessed: 2025-01-19. 2025.

[18] Ken Fisher. *OiNK?S New Piglets Proof Positive That Big Content?S Efforts Often Backfire*. 2007. URL: https://arstechnica.com/tech-policy/2007/11/oinks-new-piglets-proof-positive-that-big-contents-efforts-often-backfire/.

[19] Philippe Golle, Kevin Leyton-Brown, and Ilya Mironov. “Incentives for Sharing in Peer-to-Peer Networks”. In: *Proceedings of the 3rd ACM Conference on Electronic Commerce*. EC ’01. New York, NY, USA: Association for Computing Machinery, Oct. 2001, pp. 264–267. ISBN: 978-1-58113-387-5. DOI: 10.1145/501158.501193.

[20] Hanna Halaburda, Benjamin Livshits, and Aviv Yaish. *Platform Building With Fake Consumers: On Double Dippers and Airdrop Farmers*. en. Rochester, NY, July 2025. DOI: 10.2139/ssrn.5364583.

[21] David Hales, Rameez Rahman, Boxun Zhang, Michel Meulpolder, and Johan Pouwelse. “BitTorrent or BitCrunch: Evidence of a Credit Squeeze in BitTorrent?” In: *2009 18th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises*. ISSN: 1524-4547. June 2009, pp. 99–104. DOI: 10.1109/WETICE.2009.22.

[22] Don Johnson, Alfred Menezes, and Scott A. Vanstone. “The Elliptic Curve Digital Signature Algorithm (ECDSA)”. In: *Int. J. Inf. Sec.* 1.1 (2001), pp. 36–63. DOI: 10.1007/S102070100002. URL: https://doi.org/10.1007/s102070100002.

[23] Ian A. Kash, John K. Lai, Haoqi Zhang, and Aviv Zohar. “Economics of BitTorrent communities”. In: *Proceedings of the 21st international conference on World Wide Web*. WWW ’12. New York, NY, USA: Association for Computing Machinery, Apr. 2012, pp. 221–230. ISBN: 9781450312295. DOI: 10.1145/2187836.2187867.

[24] Patrick Tser Jern Kon, Sina Kamali, Jinyu Pei, Diogo Barradas, Ang Chen, Micah Sherr, and Moti Yung. “SpotProxy: Rediscovering the Cloud for Censorship Circumvention”. In: *33rd USENIX Security Symposium (USENIX Security*

24). Philadelphia, PA: USENIX Association, Aug. 2024, pp. 2653–2670. ISBN: 978-1-939133-44-1. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/kon.

[25] Dave Levin, Katrina LaCurts, Neil Spring, and Bobby Bhattacharjee. “Bittorrent Is an Auction: Analyzing and Improving Bittorrent’s Incentives”. In: *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*. SIGCOMM ’08. New York, NY, USA: Association for Computing Machinery, Aug. 2008, pp. 243–254. ISBN: 978-1-60558-175-0. DOI: 10.1145/1402958.1402987.

[26] Yunpeng Li, Costas A. Courcoubetis, Lingjie Duan, and Richard Weber. “Optimal Pricing for Peer-to-Peer Sharing With Network Externalities”. In: *IEEE/ACM Transactions on Networking* 29.1 (Feb. 2021), pp. 148–161. ISSN: 1558-2566. DOI: 10.1109/TNET.2020.3029398.

[27] Deepak Maram, Harjasleen Malvai, Fan Zhang, Nerla Jean-Louis, Alexander Frolov, Tyler Kell, Tyrone Lobban, Christine Moy, Ari Juels, and Andrew Miller. “CanDID: Can-Do Decentralized Identity with Legacy Compatibility, Sybil-Resistance, and Accountability”. In: *2021 IEEE Symposium on Security and Privacy (SP)*. May 2021, pp. 1348–1366. DOI: 10.1109/SP40001.2021.00038.

[28] Petar Maymounkov and David Mazières. “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric”. en. In: *Peer-to-Peer Systems*. Ed. by Peter Druschel, Frans Kaashoek, and Antony Rowstron. Vol. 2429. Berlin, Heidelberg: Springer, Oct. 2002, pp. 53–65. ISBN: 9783540457480. DOI: 10.1007/3-540-45748-8_5.

[29] Antonio Minniti and Cecilia Vergari. “Turning Piracy into Profits: a Theoretical Investigation”. In: *Information Economics and Policy* 22.4 (Dec. 2010), pp. 379–390. ISSN: 0167-6245. DOI: 10.1016/j.infoecopol.2010.06.001.

[30] Puja Ohlhaver, E. Glen Weyl, and Vitalik Buterin. *Decentralized Society: Finding Web3’s Soul*. SSRN Scholarly Paper. Rochester, NY, May 2022. DOI: 10.2139/ssrn.4105763. Social Science Research Network: 4105763.

[31] Phala Network. *Phala Network*. https://phala.com/. Accessed: 2025-01-19. 2025.

[32] R. Rahman, M. Meulpolder, D. Hales, J. Pouwelse, D. Epema, and H. Sips. “Improving Efficiency and Fairness in P2P Systems with Effort-Based Incentives”. In: *2010 IEEE International Conference on Communications*. ISSN: 1938-1883. May 2010, pp. 1–5. DOI: 10.1109/ICC.2010.5502544.

[33] RED Interview Team. *Interview for RED*. https://interviewfor.red/en/index.html. Accessed: 2025-01-19. 2025.

[34] Maurice Shih, Michael Rosenberg, Hari Kailad, and Ian Miers. *zk-promises: Anonymous Moderation, Reputation, and Blocking from Anonymous Credentials with Callbacks*. Publication info: Published elsewhere. Major revision. Usenix Security 2025. Aug. 2024. URL: https://ia.cr/2024/1260.

[35] UniRep. *UniRep Docs*. en. 2025. URL: https://developer.unirep.io/.

[36] Ben Westhoff. *Trent Reznor and Saul Williams Discuss Their New Collaboration, Mourn OiNK*. Oct. 2007. URL: https://www.vulture.com/2007/10/trent%5C_reznor%5C_and%5C_saul%5C_williams.html.

[37] Fang Wu and Li Zhang. “Proportional Response Dynamics Leads to Market Equilibrium”. In: *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*. STOC ’07. New York, NY, USA: Association for Computing Machinery, June 2007, pp. 354–363. ISBN: 978-1-59593-631-8. DOI: 10.1145/1250790.1250844.

[38] Aviv Yaish, Nir Chemaya, Lin William Cong, and Dahlia Malkhi. *Inequality in the Age of Pseudonymity*. Aug. 2025. DOI: 10.48550/arXiv.2508.04668.

[39] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. “Town Crier: An Authenticated Data Feed for Smart Contracts”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 270–282. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978326.

[40] Aviv Zohar and Jeffrey S. Rosenschein. “Adding incentives to file-sharing systems”. In: *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*. AAMAS ’09. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, May 2009, pp. 859–866. ISBN: 9780981738178. URL: https://dl.acm.org/doi/10.5555/1558109.1558131.