Accountability and Transparency for Privacy-First Network Services

Arden Ma
Stanford University

Thea Rossman
Stanford University

Abe Rosloff
Stanford University

Abstract

We present a transparent, distributed logging utility implemented in Rust and designed for a distributed network directory service based on Tor [6]. Our design uses Streamlet [5] to add application-submitted data to an internal blockchain and pushes regular "checkpoints" to a public forum. We argue that, if the network directory system is compromised, a publicly-verifiable, append-only log forces continued consistent state. We evaluate our proof-of-concept implementation within a simulated network directory application, demonstrating that the utility meets practical complexity and latency requirements similar to Tor's directory system.

1 Introduction

The Tor [3] network provides anonymous, geographically-obfuscated access to the Web. Clients select Onion Routers [6] to relay encrypted traffic: relays mask the true source and destination of a connection, allowing users to circumvent many surveillance, tracking, and censorship mechanisms.

Tor's root of trust is a small set of *directory authorities*: each authority regularly gathers network state, exchanges it with other authorities, and endorses it with cryptographic signatures. Clients independently fetch network directories from authorities, which they use to choose paths through the network. Clients trust fetched network state if and only if they can verify the signatures of a majority of authorities [2].

Our project is inspired by a "worst-case scenario": if a majority of directory servers are compromised, they could "agree on" and send malicious, arbitrary network state to users. In particular, it would be trivial for such an attacker to target specific users, maintaining normal operation for others to avoid detection.

We describe and prototype an append-only, transparent logging service to force consistency in scenarios of total network directory compromise. We propose that directory authorities create and publish a distributed, immutable record of validated network state, which clients can verify both real-time and retroactively. This system creates a layer of public accountability for any published state: it enforces network history and eliminates the ability to obfuscate an attack by targeting specific users. In other words, it would not prevent a compromised system from "lying" about network state, but it would force the system to "lie to everyone in the same way" – with a public record of doing so.

In the remainder of this paper, we describe the design and prototype implementation of our core service. We first offer additional background on our imagined use case and threat model. We briefly describe Streamlet [5], the Byzantine-Fault-Tolerant consensus protocol we use to append data to an internal log. We then describe our utility, including its API and key design. We conclude by evaluating our proof-of-concept implementation, tested with a basic application designed to behave like Tor's directory service [2], and discussing future work. We argue that our service performs at a similar time-scale to Tor's directory system, and it could be optimized much further.

2 Background

2.1 Tor's Directory Service

Our use case is modeled after Tor's directory service and how clients use it. To provide context for our design, we briefly describe this application here. [2]

The Tor network has 10 trusted directory authorities[1]. Each directory authority regularly builds a *local* view of the Tor network state ("directory") based on advertisements from relays; every hour, authorities initiate a consensus protocol to collectively sign a single directory. Each network state contains a "router descriptor" from each of the 6,000+ relays on the Tor network; each includes, e.g., the relay's IP and port, exit policies, public key, bandwidth metrics, and published date.[2]

From a client perspective, any directory signed by a majority of directory servers is a trusted representation of relays on the network. (Clients are shipped with authorities' identities and public keys.) Clients choose relays to forward traffic through, both randomly and prioritized based on proximity, bandwidth, longevity in the network, exit policies, and other metrics. [4]

For our design and threat model, we note six observations about Tor's directory service:

- Tor uses a well-established, relatively complex protocol to generate network state; for deployability, we propose to *add* to, rather than *replace*, it.
- Network state reconciliation occurs every hour, suggesting, for our system, infrequent updates.
- The data we wish to append to a log is relatively large: 400-500KB per directory.
- With 10 authorities and a majority-vote system, compromise of just 6 servers leads to full system compromise.
- Directory authorities service client requests independently; without a public log, nothing forces consistent state.
- Given a manipulated directory containing significant malicious relays, perhaps with inflated metrics, a client could easily build a circuit of entirely compromised nodes.

2.2 Threat Model

The key adversary we design for is a committed, well-resourced attacker who aims to compromise the anonymity and integrity of Tor (or a similar service). The attacker is able to compromise six out of ten directory authorities, signing and sending arbitrary state and inducing clients to forward traffic through malicious relays. To evade detection, the attacker targets specific clients while maintaining normal operation for others.

We also consider an adversary able to compromise *our* logging utility. We use a Byzantine-Fault-Tolerant protocol that maintains liveness and integrity if at most f out of $N = 2f + 1$ nodes are compromised. If $> f$ nodes are compromised, we prefer denial-of-service to undetectable compromise.

In each of these scenarios, our system aims to force *transparency and consistency*. An attacker should be forced to share consistent state with all users. And, an attack should be retroactively transparent: in other words, attackers should not be able to send malicious information to a user, that will be accepted by that user, without a public record of doing so.

2.3 Streamlet

The nodes in our system use partially-synchronous Streamlet [5] to come to consensus on a growing log, stored as a *closed-membership blockchain*. We briefly describe Streamlet here.

Streamlet takes application data as "transactions" and outputs a unique, finalized blockchain encapsulating these transactions. Along with data, each block includes a hash of the parent block, the epoch in which the block was proposed and voted on, and data. Note that, assuming a collision-resistant hash, each block implies the blocks that preceded it.

Streamlet nodes proceed in synchronized epochs and use a propose-vote system to append blocks. In epoch e, each node independently computes the "epoch leader" based on a deterministic hash of e. The leader proposes a block that extends from a longest notarized chain (see below) and encapsulates transactions received by the application. Remaining nodes vote for the block if and only if (1) it originates from the epoch leader, (2) it extends a longest notarized chain, and (3) the node has not yet voted in epoch e. In our system, nodes also vote for blocks if and only if the encapsulated transactions conform to an application-specific "contract". Proposals and votes are cryptographic signatures on the data.

Nodes broadcast proposals and votes to all other nodes. Further, each node forwards all new messages received to all other nodes. This "implicit echoing assumption" significantly increases network load, but it offers the protocol stronger guarantees and provability.

[1]https://en.wikipedia.org/wiki/Tor_(network); https://blog.torproject.org/introducing-bastet-our-new-directory-authority/

[2]https://metrics.torproject.org/collector.html

Each instance considers a block "notarized" when signed by at least $2N/3$ out of N nodes. On observing three adjacent blocks in a notarized blockchain with consecutive epoch numbers, a node finalizes the second of three blocks, along with its prefix chain. Streamlet guarantees liveness (continued operation) and integrity (all functioning nodes agree on finalized blocks) in the presence of $\geq 2N/3$ honest and functioning nodes.

Note that our use of Streamlet implies three assumptions. First, we guarantee integrity and liveness if and only if, for a group of $N = 2f + 1$ nodes, up to f nodes are corrupted. We assume that corrupted notes may exhibit arbitrary behavior (Byzantine faults). Second, we assume a partially synchronous setting. Given that the network we design for is the global Internet, we believe this to be reasonable with a well-tuned RTT estimate. Finally, to guarantee liveness, we require clocks to have bounded drift significantly smaller than epoch time (5.2).

3 System Design

3.1 Overview and API

A directory application interacts with our utility in three ways: (1) **Propose** data to append, (2) **Request** the most recent **finalized data**, and (3) **Request** the entire **finalized log**. Using Streamlet, the utility guarantees that *finalized* directories are consistent across functioning nodes and appended to the internal append-only log (blockchain). Clients **Request** finalized blocks and chains from logging nodes to validate data sent by the directory service.

When initialized, authorities would negotiate a mutual validation scheme with the logging service (e.g., signature checks). Similarly, clients would need to be shipped with the capacity to validate log data (e.g., public keys).

Finally, our utility publishes "checkpoints" – either the most recently finalized block or its hash – on a configurable interval to a public, append-only log. Assuming a collision-resistant hash, this "checkpoint" uniquely determines its entire prefix chain: if our service were compromised, it could not retroactively modify the log before the last checkpoint without detection. In our proof-of-concept implementation, this "log" is simply a shared file, and we push an entire block. In the future, we imagine this as a public blockchain (5.7). Clients require an additional API to access and validate public logs, which we omit from our initial implementation.

3.2 Components

Internally, we implement four key modules: Blocks, Chains, and Blockchain Management; Messages; Network Stack; and Streamlet Instances.

Each **Block** contains the epoch in which it was proposed and voted on, a hash of the block, a hash of its parent block, and raw data. Blocks are appended to **Chain**s, accepted if they correctly extend the chain. The **Blockchain Manager** tracks and finalizes longest notarized chains.

Nodes send and receive **Messages** – `Propose`, `Vote`, and message types for communicating with the application.

Our **Network Stack** triggers events when messages are received and broadcasts messages when requested by Streamlet. In our implementation, the network stack wraps a `LibP2P` open-source flooding protocol [1, 10] over a virtual local network and encrypted transport layer. For sending larger messages to the application, we use point-to-point messages.

The majority of our logic is in the **StreamletInstance**. Each instance tracks epochs, sends proposals and votes, determines thresholds for notarization, logs finalized data, and exchanges messages with the application. The `StreamletInstance` is event-driven: an epoch timer triggers leader selection and proposals, and `NetworkStack` events trigger echoing and votes. Instances are initialized with the public keys of peers. Each instance encapsulates a `BlockchainManager`, which it forwards notarized blocks to and retrieves finalized chains from.

4 Evaluation

We now present and evaluate our proof-of-concept implementation of our Streamlet-based logging utility. The code is available on GitHub.[3].

We evaluate N Streamlet nodes running on a single machine (though the implementation would also work across several machines on a virtual local network). A brief initialization period loads each node with the others' IP addresses and public keys. In a separate process, an application generates network directories, modeled off of Tor's [2], flooding this data to all Streamlet

[3]https://github.com/ardenma/streamlet-transparency-logging

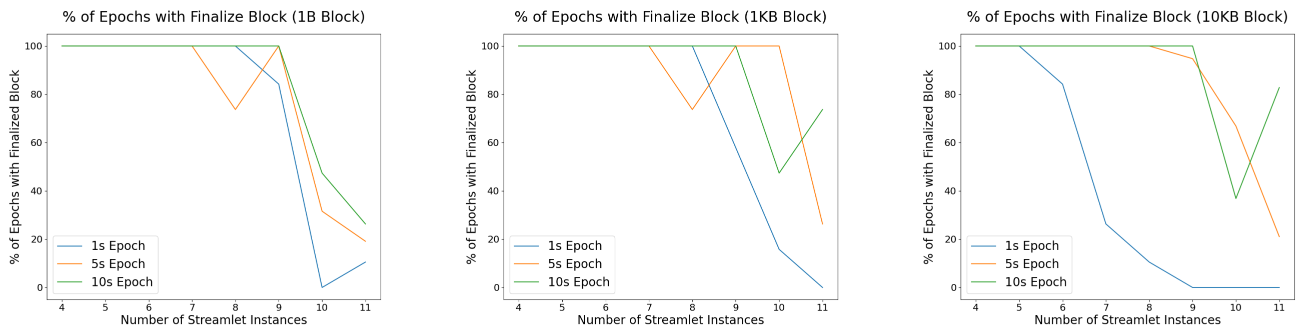

Figure 1: Epochs that were able to finalize blocks, as a percentage of optimal, based on data size, epoch length, and number of nodes. Longer epoch times, smaller data, and fewer nodes lead to more consistent finalization.

nodes. Nodes run Streamlet, over a `LibP2P`-based network stack [1]. We model the "public log" as a shared file, which nodes append to at each epoch.

4.1 Performance: Finalization

The core metric we consider is: given an epoch time, number of nodes, block size, and available transaction data, in what fraction of epochs is our log able to finalize blocks? Since the network directory application cannot share data with a client until it is finalized on the log, finalization is critical to the performance of the application. In other words, we consider latency and throughput a function of epoch length, and we aim to make our epoch length as short as possible.

We measure the epochs that were able to finalize blocks as a percentage of optimal. For X epochs, $X-1$ is optimal: finalization can begin at the second epoch, when there are at most three blocks in the chain (the genesis block with epoch 0, the block appended at epoch 1, and the block appended at epoch 2). If each subsequent epoch successfully notarizes a block, then $X-1$ epochs are able to finalize blocks.

Figure 1 shows the results of running these skews on a single m6a.4xlarge on AWS with 16 vCPUs and 64GB of memory. To test how node count affected finalization, we started each "node" in a separate process and exchanged messages using our network stack, simulating inter-machine communication. As expected, finalization becomes more challenging with shorter epochs, more nodes, and, to some extent, larger data. In cases with low finalization, we observe that blocks are still *notarized*, but message echoes continuing from epoch $i-1$ preempt proposals and votes in epoch i – preventing 3 *consecutive* epochs from notarizing blocks.

To apply for our use case, our implementation should mimic the timescale of Tor's network directory service. Testing our proof-of-concept implementation locally, we were able to achieve near-perfect block-per-epoch finalization of a realistic network directory (500KB) across 8 streamlet nodes in 20 minutes by using 25-second epochs and appending 10KB chunks. This throughput is comparable to Tor's timescale – and, with the optimizations described in 3.2 and more CPU/RAM, has significant potential to be improved.

4.2 Compromised Node Handling

We tested Streamlet's performance with $< 1/3$ compromised or malfunctioning nodes. We ran our implementation with 4 nodes – a scenario that offered consistently perfect performance in the fully-functional case – and set one node to behave maliciously. This node sent invalid blocks, crashed, became out of sync with other nodes' epochs, or failed to broadcast votes and proposals.

Scenarios affecting the *voting* process presented no impact: nodes that failed to vote, sent invalid votes, sent proposals as non-leaders, or failed to broadcast had no impact. Situations in which the *leader* failed to propose, whether due to crash or malicious behavior, decreased finalization. As expected, the degree of performance impact depended on leader selection: if the malicious node is chosen as leader every third epoch, Streamlet becomes unable to finalize blocks (5.3). Our examination shows that our design is resistant to compromised-voting/wrongfully proposing nodes and was able to seamlessly handle invalid message data.

5 Discussion and Future Work

5.1 Logging-as-a-Service

We chose to build our logging utility *separate* from Tor's directory consensus process, rather than as part of it. As Hunt et. al. argue in [8], an independent service can scale independently of the application to tune availability and safety. We considered deployability and maintenance: adding functionality alongside a well-established system is easier than modifying it. And, maintaining a conceptual focus on a specific service mimics a control/data plane separation (see, e.g., Aurora [9] and GFS [7]).

However, this separation means that our system adds latency. In other words, appending occurs *after* consensus on network state. To remove this overhead, one could imagine a directory service that runs consensus on network state *and* appends to a log as part of the same protocol. We leave rigorous discussion of this tradeoff to future work.

5.2 Message Size and Communication Complexity

A key limitation of Streamlet is the complexity of network communication: even under perfect operation, in an optimized implementation, notarizing a single block requires at least $O(N^2)$ messages for a system of N nodes. For each block, (1) a leader broadcasts a proposal to all nodes ($O(N)$ messages; $O(N\cdot \texttt{blocksize})$ bits); (2) each node "echoes" this proposal to each other node ($O(N^2)$ messages; $O(N^2\cdot \texttt{blocksize})$ bits); (3) each node votes on the block and broadcasts its vote to all other nodes ($O(N)$ messages; $O(N \cdot \texttt{votesize})$ bits), and (4) each node forwards each vote it receives to each other node ($O(N^2)$ messages; $O(N^2 \cdot \texttt{votesize})$ bits). While the number of messages is unavoidable in Streamlet, smaller messages can help.

Note that our gossip protocol also imposes a message size limit. In our proof-of-concept, we limit the block size submitted by the application; it appends one realistic Tor directory across 20 transactions. To exchange chains, we use point-to-point messages over raw TCP sockets.

As future work, we suggest intentionally decreasing message size. Aside from using a different consensus protocol, we imagine two approaches to this: changing how Streamlet stores data or changing how the directory shares data.

First, we considered, but were not able to implement, a collision-resistant hash solution: each Streamlet node could receive an identical directory from the application, independently compute the block to append, and exchange the block metadata – block hash, parent hash, epoch number, etc., without flooding and echoing directory data. Nodes locally validate proposals and votes. Larger data could still be exchanged, in a different thread, over point-to-point messages.

Second, one could imagine the application sending data more similar to Tor's directory caching specification [2]. In this, directories are sent in the form of "updates" – i.e., changes from the last directory – rather than full directories. Adopting this compressed data format is likely to considerably cut down on size.

5.3 Clock Synchronization

We considered a rigorous timing protocol to be beyond the scope of our project. In our proof-of-concept, we synchronize epochs by assuming relatively long epoch times, bounded and predictable clock drift, and relatively short uptime. We recommend two alternatives for future work: either a timing protocol to drive epoch transitions (e.g., Tor finds NTP to be sufficient for their epoch times), or migration to a consensus protocol that does not rely on synchronized epochs (e.g., HotStuff [12]).

5.4 Finalization Requirements

When f malicious leaders are introduced, Streamlet may become unable to consistently finalize blocks. This becomes less likely with more nodes; however, more nodes add communication complexity that, in turn, leads to less reliable finalization (5.1, 4). If we consider practical "liveness" for an application to be consistent block finalization, our system, as-is, cannot guarantee this "liveness" when nodes malfunction.

We chose Streamlet as a first step for our proof-of-concept: it is intentionally simple, easy to prove, and pedagogical. A true deployment using Streamlet would require regular monitoring of the log: if the service is consistently unable to finalize, it must be examined incorrect or compromised behavior. A next-step implementation would replace Streamlet with a more practically robust protocol, e.g., HotStuff [12].

5.5 Network Stack

Our network stack wraps a `LibP2P` [1] library. We chose `LibP2P` because it offers an open-source implementation in Rust and an event-driven interface, we were able to use a gossip protocol [10] that floods and echoes messages as required by Streamlet, and the library was developed for blockchain and other peer-to-peer consensus applications.

Unfortunately, as we stress-tested our implementation, we identified an issue in `libp2p`: as network load, number of peers, and duration of uptime increased, connections became less reliable. Messages became more delayed, and peers became more likely to disconnect and reconnect, drop packets, or leave the network altogether. We believe that this stems from two issues. First, the protocol implementation we use – across all examples we tested – requires significant processing power and memory. Second, we identified a bug in `libp2p`'s peer maintenance protocol, which caused peers to erroneously time out; we submitted a report to the maintainers[4] and intend to work on a fix after the quarter ends.

As long as $2N/3$ nodes remain operational and connected, Streamlet continues to notarize and regularly finalize blocks, though we do observe added latency and retransmission (e.g., if an epoch leader temporarily goes offline) and, if a node misses an epoch altogether, it is unable to catch up (5.5).

As a whole, we believe that a network stack based around `libp2p` may impose a bottleneck and unintentional unreliability into our system.

5.6 Recovery and View Changes

We do not implement recovering a node from a crash or, in some cases, an entirely missed epoch. The Streamlet specification did not describe reliable and correct recovery, though we assume implementing this would involve a logging-and-reading approach. We also did not implement adding or removing nodes from Streamlet, which we argue is not common to our use case: because clients must be shipped with a key for each nodes, it is challenging and rare to add an additional node following deployment.

[4] https://github.com/libp2p/rust-libp2p/issues/2676

5.7 Smart Contracts and a Public Log

Our system assumes the a secure public log, where data can be posted by our system, be queried by users, and remain immutable. In our proof-of-concept, this "log" is a file. In a rigorous implementation, this could be a public blockchain (e.g., Ethereum [11]), though a Certificate Transparency system may also be feasible and lower-cost.[5] The desired behavior could be achieved with a smart contract, initialized with the public keys of each Streamlet node. The log should only accept data with a threshold of signatures, and clients should expect finalized data to be published regularly.

6 Conclusion

We describe and implement a distributed, transparent, and reliable logging service designed for a privacy-first, distributed network directory system modeled off of Tor. Our implementation uses Streamlet to append network directory data to an internal, permissioned blockchain; pushes auditable "checkpoints" to a public log; and shares finalized data with both clients and the directory application. We show that our proof-of-concept implementation can operate at a time-scale similar to Tor's, and we describe optimizations that, we believe, could reduce added latency significantly. We argue that an append-only, publicly-auditable logging system adds protection in the case of total directory compromise: while it would not prevent the directory from manipulating network state, it would force global consistency, transparency, and retroactive auditability.

References

[1] Libp2p: An open source project from the ipfs community. https://libp2p.io. Accessed: 2022-05-01.

[2] Tor directory protocol, version 3. https://gitweb.torproject.org/torspec.git/tree/dir-spec.txt. Accessed: 2022-05-01.

[3] The tor project. https://www.torproject.org. Accessed: 2022-05-01.

[5] Mozilla publishes hashes of binary releases to Certificate Transparency logs: https://wiki.mozilla.org/Security/Binary_Transparency.

[4] Tor protocol specification. https://github.com/torproject/torspec/blob/main/tor-spec.txt. Accessed: 2022-05-01.

[5] Benjamin Chan and Elaine Shi. Streamlet: Textbook streamlined blockchain. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, 2020.

[6] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Naval Research Lab*, 2004.

[7] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 20–43, Bolton Landing, NY, 2003.

[8] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 11, USA, 2010. USENIX Association.

[9] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1041–1052, New York, NY, USA, 2017. Association for Computing Machinery.

[10] Dimitris Vyzovitis, Yusef Napora, Dirk McCormick, David Dias, and Yiannis Psaras. Gossipsub: Attack-resilient message propagation in the filecoin and eth2.0 networks. In *Computing Research Repository (CoRR)*, 2020.

[11] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger.

[12] Dahlia Yin, Maofan an Malkhi, Michael Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus in the lens of a blockchain. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019.