Scalable Log Auditing on Private Blockchains via Lightweight Log-Fork Prevention

Yuzhe Tang
ytang100@syr.edu
Syracuse University

Kai Li
kli111@syr.edu
Syracuse University

Yibo Wang
ywang349@syr.edu
Syracuse University

Sencer Burak Somuncuoglu*
burak.somuncuoglu@chainalysis.com
Chainalysis

Abstract

This work presents TxChecker, a secure logging system over a private blockchain with two salient features: 1) TxChecker prevents log forking attacks without trusting any external party other than the blockchain, 2) TxChecker achieves a low cost on log auditors that is proportional to the data being audited. TxChecker employs a novel scheme to map the misbehavior of forking logs to the double-spending transactions, which are invalidated by the underlying blockchain. In the TxChecker protocol, clients and the server in the domain infrastructure both attest to a history of concurrent operations and send blockchain transactions. A prototype of TxChecker is implemented in HyperLedger Fabric. Evaluation shows that TxChecker reduces the costs of log auditors significantly compared to replication-based log schemes.

CCS Concepts: • **Security and privacy → Distributed systems security**.

Keywords: Blockchains, auditable logs, light clients.

ACM Reference Format:
Yuzhe Tang, Kai Li, Yibo Wang, and Sencer Burak Somuncuoglu. 2020. Scalable Log Auditing on Private Blockchains via Lightweight Log-Fork Prevention . In *Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers (SERIAL'20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 4 pages.
https://doi.org/10.1145/3429884.3430032

1 Introduction

Preventing log forks among disconnected auditors, such as web browsers, is known to be impossible without trusted-third parties [14]. A promising solution is to use blockchains as a trusted source of fork prevention, as in recently proposed blockchain logs such as Catena [17], Chainiac [15] and Ghostor [9]. While the log actors in these schemes avoid downloading a blockchain full node by using "light" clients such as simplified payment verification (SPV) clients [3, 5], they still incur high auditing overhead caused by the replicated full logs across auditors. The high auditing overhead presents a major bottleneck to scale the log auditing to a large population. This work aims at lightweight prevention of log forks for scalable auditing – Given a log of N entries, an auditor on a single entry can be assured of no log forks by downloading $O(N)$ log data (and blockchain data). We call this property by *distributed log auditing* or DLA.

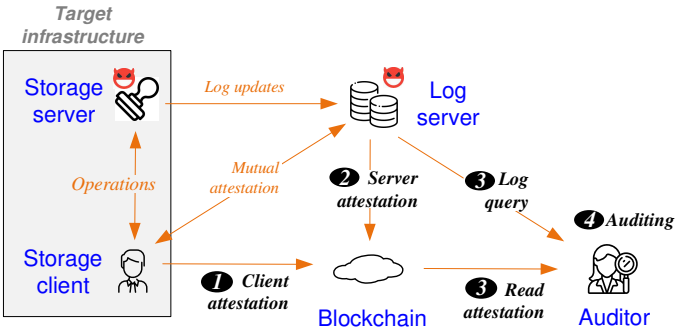


Figure 1. TxChecker system model

We propose TxChecker, a log running over blockchain's SPV client and achieving DLA. In TxChecker, the overall system in Figure 1 consists of a private blockchain and four actors: a storage server, storage clients, a log server which may co-reside with the storage server, log auditors who may co-reside with storage clients. Storage clients submit data read/write operations to the storage server which process the operations concurrently. The private blockchain stores its ledger state in unspent transaction outputs or UTXOs [4]. In this work, a private blockchain are chosen because it is more efficient than a public blockchain and is better suited to observe the intensive operation stream. Also, a private blockchain that federates a consortium of trust domains is more resilient against insiders or external hacks than a conventional database in a single domain.

*Work done when the author is a master student at Syracuse University.

SERIAL'20, December 7–11, 2020, Delft, Netherlands

ACM ISBN 978-1-4503-8208-3/20/12…$15.00
https://doi.org/10.1145/3429884.3430032

In the threat model, the storage server can misbehave and the adversarial log server can mount forking attacks to present different views to different auditors. The blockchain is trusted to honestly validate transactions and provide immutability; a double-spending transaction cannot be validated, and once be added to the blockchain, a transaction cannot be altered.

Unlike existing log schemes [17] where only the log server sends log attestations to the blockchains, TxChecker makes both the server and clients send log attestations to the blockchain. The log stores a history of storage operations, linearized in an order consistent to their real-time relations. An auditor on operation o downloads only the predecessor and successor of the operation in the linearized log and can be assured no forks occurred. The security (of no log forks) can be reduced to the blockchain security assumption that no double-spends are validated and the assumption that trusted clients will not replay their log attestations.

As an application, TxChecker can verify the consistency of concurrent storage operations against an untrusted server. The consistency model is linearizability [8]. A TxChecker auditor verify that concurrent operations belong to a linearizable history.

We build a system prototype of TxChecker on FabToken [4], which is a UTXO-based token contract on HyperLedger Fabric [2].[1] We evaluate the client cost of TxChecker in comparison to the existing replication-based log schemes. We build a non-trivial cost model by counting the time duration and size of the data a log auditor is required to maintain. The target metric affecting auditing costs is blockchain's transaction finality time. We conduct experiments by driving the standard YCSB workloads [6] to the system and measure the transaction finality delays of our TxChecker prototype on FabToken. The experiment results show that with more than hundreds of clients, TxChecker saves the client cost by one to two orders of magnitude, compared with replicated log schemes e.g., Catena.

2 Related Works

Among **blockchain applications** to security infrastructures, IKP [13] leverages cryptocurrencies to incentivize the reporting of mis-issued certificates in PKI and can be complementary to TxChecker. ContractChecker [11] conducts storage consistency verification in smart contracts. It adopts the on-chain design where log auditing occurs fully on the blockchain, thus incurring high costs in smart-contract execution. GEM^2 trees [18] and TPAD [16] support the secure database outsourcing to the hybrid blockchain-cloud platform, enabling authenticated database queries at low blockchain costs. These systems assume a single log auditor/data user and do not address the fork prevention as in TxChecker. The design of **blockchain light clients** [3, 5, 10] is to avoid downloading a full transaction history on a blockchain client. The communication efficiency and reliability between the blockchain network and a particular full-node client is addressed in [7]. These techniques are complementary to this work which uniquely addresses the design of a light client for blockchain logs.

3 The TxChecker Protocol

This section presents the TxChecker protocol. Due to the space limits, the full security analysis is deferred to technical report [1]. The protocol runs in epochs. In each epoch, there are three steps: Step ❶ occurs with each storage operation a client sends to the storage server. By the end of an epoch, the server attests to the batch of operations (Step ❷), serves log queries from individual auditors (Step ❸ & ❹).

Initially, there is a "genesis" transaction for each data key used. The genesis transaction is sent by an offline trusted party (e.g., a client). The transaction identity is propagated to all participating clients and the server. We assume all participating clients are available in this initialization process.

Step ❶: Client log attestation: Client c attests to her operation o by calling $tx_c[o]$ = `clientAttest`(c, o). Specifically, suppose a client c sends a request of operation o to the server at time $o.inv$. The server processes the requests and sends the response at time $o.rsp$. Here, client c signs her request, and the server signs the response, making the operation double-signed. The client generates a transaction $tx_c[o]$ that encodes the double-signed operation o (e.g., using the `OP_RETURN` instruction). The transaction transfers a fixed amount of cryptocurrency coins from the client c's address to the server's address. The transaction is signed by the private key controlled in client c's wallet. The client will send the ID of transaction $tx_c[o]$ to the server.

Step ❷: Server log attestation: At the end of every epoch, the server collects all the operations processed and, for each key, declares a total-order on the operations of the key. To find the total-order, the server can run existing linearizability checking algorithms [12]. The server then attests to the total-order by generating a series of server transactions. Formally, given operations $\{o\}$ and their client transactions $\{tx_c[o]\}$, the server attests to the total order by generating server transactions in order: $\{\langle o, tx_c, tx_s\rangle\}$ = `serverAttest`$(\{\langle o, tx_c\rangle\})$. For each operation in the total-order, the server tags a value called min_rsp, which is the minimal response time, the next operation is allowed to have. Specifically, suppose operation o' is immediately succeeded by o. The server tags operation o with the min_rsp value calculated as follows:

$$min\_rsp[o] = \max\left(min\_rsp[o'], o.inv\right) \quad (1)$$

[1] Note that we dismiss the alternative design to support TxChecker on account-based blockchains directly (e.g., native Ethereum) due to the lack of light-client supports there.

The server generates a transaction to encode operation o and $min\_rsp[o]$, that is, $tx_s[o\|min\_rsp[o]]$. Transaction $tx_s[o]$[2] fully spends the outputs of two transactions, $tx_c[o]$ and $tx_s[o']$. Note that the previous operation o' may be processed in a previous epoch. If operation o is the very first operation of a given key, the server transaction $tx_s[o]$ will spend the output of the genesis transaction for the key $tx_g(o.k)$, and the output of client transaction $tx_c[o]$. If operation o is a write $w(k, v)$, the server transaction $tx_s[o]$ encodes the key and value of the record written. If the operation o is a read $v = r(k)$, the server transaction $tx_s[o]$ also encodes both the key of the read (k) and the value of the record read (v).

Step ❸: Submitting log query: A client who sent operation o is interested in auditing the consistency of the operation in the current total-order log. To do so, the client poses a log query of operation o on the current epoch's total-order log. Given a log digest dg and a query on operation o, the server processes the query against the log by returning operation o and its preceding operation o', associated with their server/client transactions: $\langle o, tx_c[o], tx_s[o]\rangle$, $\langle o', tx_c[o'], tx_s[o']\rangle$ = `serverLogQuery`(o, dg). If o is the first operation by itself, the second triplet in the result is $\langle tx_g(o.k)\rangle$.

Step ❹: Log auditing based on query results: The result of log query constitutes the view of this client, that is, $\langle o, tx_s[o], tx_c[o]\rangle$, and $\langle o', tx_s[o'], tx_c[o']\rangle$. Based on the view, the client conducts the audits: b = `clientLogAudit`$(c, \langle o, tx_c[o], tx_s[o]\rangle, \langle o', tx_c[o'], tx_s[o']\rangle, o)$. The audits include the following two steps:

Step 4-1) Log integrity checks: The client checks the authenticity of the predecessor operation presented by the untrusted server. The purpose of the authenticity check is to detect and prevent the server forging and forking the total-order log. **4-1a)**: The client transaction's signature can be verified against its public key. That is, the signature of $tx_c[o']$ can be verified by client $c(o')$'s public key. Here, we assume clients are identified and their public keys are securely distributed among all clients. **4-1b)**: Server transactions fully spend the outputs of the corresponding client transactions. Specifically, the output of $tx_c[o]$ ($tx_c[o']$) is fully spent by $tx_s[o]$ ($tx_s[o']$). **4-1c)**: The transaction $tx_s[o]$ fully spends the output of transaction $tx_s[o']$ or $tx_g(o.k)$. **4-1d)**: All transactions in the view, i.e., $tx_c[o]$, $tx_s[o]$, $tx_c[o']$ and $tx_s[o']$ (or $tx_g(o.k)$), are finalized n the blockchain.

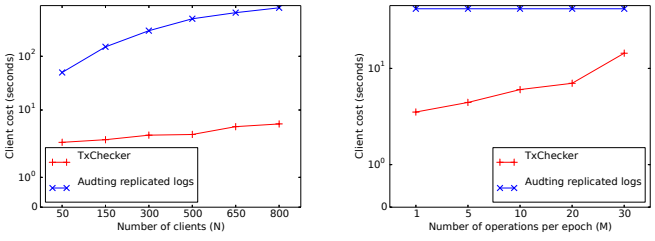
Step 4-2) Consistency checks: The client checks the consistency on her view of operation history and ensure the linearizability [8]. Informally, given a series of concurrent operations, linearizability states that the operations can be serialized to a total-order without violating the causal/real-time order among them. Linearizability definition can be found in [8]. **4-2a)**: Check real-time order: Operation o does not happen before operation o' or any earlier operations. The following conditions are checked: $min\_rsp[o] = max($ $min\_rsp[o']$, $o.inv)$ (the same as in Equation 1) and $min\_rsp[o'] < o.inv$. **4-2b)**: Check read-write freshness: If o is a read operation, say $v = r(k)$, and o' is a write, say $w(k', v')$, check $k' = k$ and $v = v'$. If both o and o' are reads, say $v = r(k)$ and $v' = r'(k')$, check $k = k'$ and $v = v'$. If both o and o' are writes, say $w(k, v)$ and $w'(k', v')$, check $k = k'$.

Prototype impl.: We build a TxChecker prototype on FabToken. FabToken [4] is a UTXO-based contract that runs over the permissioned blockchain of HyperLedger Fabric. FabToken supports token transfer and represents each transfer by a mapping between input (a token spent from the sender's address) and output (a new token spendable by the receiver's address). In other words, FabToken stores the token ownership state in the UTXO model.

With FabToken, TxChecker is implemented as a middleware over HyperLedger Fabric's off-chain clients. The middleware translates log updates into FabToken `transfer()` calls so that each log update is represented by a token. We abuse the "type" attribute in a FabToken to store the external log information. TxChecker sees HyperLedger Fabric's blockchain as a blackbox and is unaware of its internal execute-order-validate workflow. TxChecker uses default endorsement policies and may abort conflicting transactions in its order phase.

4 Evaluation

In this section, we evaluate the client cost of TxChecker in comparison with replicated log auditing schemes (RLA). We first build a client-cost model and, based on the model, perform experiments. transaction delay, denoted as $\frac{F_t}{N}$.



(a) Vary number of clients **(b)** Vary operation number

Figure 2. Client costs of TxChecker, where the client cost of replicated log is normalized and the client cost of distributed log is measured by transaction delay F_t.

Experiment design: In RLA (replicated log auditing), each auditor receives the log updates from all clients; the client cost in replicated logs is proportional to the number of clients N. In DLA (distributed log auditing), each auditor receives a constant number of log updates and needs to maintain them for an extended period of time until the transactions are finalized in the Blockchain (time period F_t). Based on this, we

[2]We use terms $tx_s[o]$ and $tx_s[o\|min\_rsp[o]]$ interchangeably.

can derive (see details in technical report [1]) that the client cost of distributed logs divided by the cost of replicated logs is proportional to transaction-finality delay F_t, which is thus chosen as the primary metric of client cost.

In experiments, we set Epoch E to be longer than F_t, so that the Blockchain will not drop transactions eventually. To measure F_t, we run N clients concurrently who submit their transactions to the Blockchain. The N clients then proactively check the finality of the transactions. We used the suggested transaction fee and did not observe any transactions being dropped. We record the time duration between the submission time of the first transaction and the finality of the last transaction on Blockchain.

Platform setup: We set up our TxChecker prototype on HyperLedger Fabric in a cloud environment. We created four servers using AWS EC2 instances, among which three act as peer nodes in the HyperLedger Fabric and one server acts as the orderer node. The hardware specs of these servers are the following: 3.0 GHz Intel Xeon Processor with two virtual CPUs, 8 GB memory, 8 GB SSD storage. All these servers run Ubuntu 16.04 LTS 64-bit, and use LevelDB as HyperLedger Fabric's state database. On the client-side, we use several laptops as client machines. The YCSB benchmark [6] is used to generate transaction proposals. Particularly, in the offline phase, we use YCSB to generate a trace of reads and writes (on workload A of 50% reads and 50% writes, with data keys following the Zipfian distribution). We then replay the same trace on client machines, in each experiment, to generate transaction proposals in TxChecker. The transaction proposals are sent, at a controlled rate, to the peer nodes in HyperLedger Fabric, such that it does not overwhelm the blockchain network, and no transactions are dropped.

Experiment results: Figure 2a shows the client cost per operation with a varying number of clients. With the increasing number of clients, the client cost of RLA[3] increases linearly (note the log scale of client cost plotted on the figure). By contrast, the client cost of TxChecker grows much slower. With 800 clients, the TxChecker cost is lower than that of RLA by two orders of magnitude. This is due to that each operation in RLA needs to be replicated on all N clients while an operation in TxChecker is replicated on two clients, leaving its cost independent with the number of clients.

Figure 2b shows the per-operation client cost with varying number of operations submitted by a client. With more operations (packed in an epoch), the client cost of TxChecker increases; because with more operations, the current block in the blockchain is more likely to be saturated, incurring a longer delay on a TxChecker auditor. The client cost of RLA stays constant when the number of operations per epoch increases. Because in RLA, a client never truncates the log, and the transaction delay does not affect the client's cost.

[3]Note that we use RLA to represent the design of Catena.

Acknowledgments

This work was supported by the National Science Foundation under Grant CNS1815814.

References

[1] Anyone can audit: Lightweight log auditing with fork prevention via blockchains. https://tinyurl.com/y5xtfmer.
[2] Hyperledger fabric, https://www.hyperledger.org/projects/fabric.
[3] Simplified payment verification: http://docs.electrum.org/en/latest/spv.html.
[4] Using FabToken on HyperLedger Fabric, http://bit.ly/357zfgg.
[5] B. Bunz, L. Kiffer, L. Luu, and M. Zamani. Flyclient: Super-Light Clients for Cryptocurrencies. 2019. https://eprint.iacr.org/2019/226.
[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In SoCC, pages 143–154, 2010.
[7] J. Hellings and M. Sadoghi. Coordination-free byzantine replication with minimal communication costs. In C. Lutz and J. C. Jung, editors, ICDT 2020, March 30-April 2, 2020, Copenhagen, Denmark, volume 155 of LIPIcs, pages 17:1–17:20, 2020.
[8] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst., 12(3):463–492, 1990.
[9] Y. Hu, S. Kumar, and R. A. Popa. Ghostor: Toward a secure data-sharing system from decentralized trust. In NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020, pages 851–877, 2020.
[10] A. Kiayias, N. Lamprou, and A. Stouka. Proofs of proofs of work with sublinear complexity. In FC 2016 Workshops, pages 61–78, 2016.
[11] K. Li, Y. Tang, B. H. B. Kim, and J. Xu. Secure consistency verification for untrusted cloud storage by public blockchains. In S. Chen, K. R. Choo, X. Fu, W. Lou, and A. Mohaisen, editors, SecureComm 2019, Orlando, FL, USA, October 23-25, 2019, Proceedings, Part I.
[12] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential consistency: measuring and understanding consistency at facebook. In SOSP 2015, pages 295–310, 2015.
[13] S. Matsumoto and R. M. Reischuk. IKP: turning a PKI around with decentralized automated incentives. In SP 2017, pages 410–426, 2017.
[14] D. Mazières and D. Shasha. Building secure file systems out of byantine storage. In PODC 2002, pages 108–117, 2002.
[15] K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, I. Khoffi, J. Cappos, and B. Ford. CHAINIAC: proactive software-update transparency via collectively signed skipchains and verified builds. In E. Kirda and T. Ristenpart, editors, USENIX Security 2017, pages 1271–1287. USENIX Association, 2017.
[16] Y. R. Tang, Z. Xing, C. Xu, J. Chen, and J. Xu. Lightweight blockchain logging for data-intensive applications. In FC 2018 International Workshops, WTSC, Nieuwpoort, Curaçao, March 2, 2018, pages 308–324, 2018.
[17] A. Tomescu and S. Devadas. Catena: Efficient Non-equivocation via Bitcoin. In SP 2017, pages 393–409, 2017.
[18] C. Zhang, C. Xu, J. Xu, Y. Tang, and B. Choi. Gemˆ2-tree: A gas-efficient structure for authenticated range queries. In ICDE 2019, pages 842–853. IEEE, 2019.