Cost-Effective Data Feeds to Blockchains via Workload-Adaptive Data Replication

Kai Li
kli111@syr.edu
Syracuse University
Syracuse, NY

Yuzhe Tang*
ytang100@syr.edu
Syracuse University
Syracuse, NY

Jiaqi Chen
jchen217@syr.edu
Syracuse University
Syracuse, NY

Zhehu Yuan†
zy2262@nyu.edu
New York University
New York, NY

Cheng Xu
chengxu@comp.hkbu.edu.hk
Hong Kong Baptist University
Kowloon, Hong Kong

Jianliang Xu
xujl@hkbu.edu.hk
Hong Kong Baptist University
Kowloon, Hong Kong

Abstract

Feeding external data to a blockchain, a.k.a. data feed, is an essential task to enable blockchain interoperability and support emerging cross-domain applications. Given the data-intensive nature of real-life feeds (e.g., high-frequency price updates) and the high cost of using blockchain, namely Gas, it is imperative to reduce the Gas cost of data feeds. Motivated by the constant-changing workloads infi nancial applications, this work aims at designing a *dynamic, workload-aware* approach for Gas cost optimization. This design space is understudied in existing blockchain research which has so far focused on static data placement.

This work presents GRuB, a cost-effective data feed that dynamically replicates data between the blockchain and off-chain cloud storage. GRuB monitors the current workload and makes data-replication decisions in a workload-adaptive fashion. Online algorithms are proposed to bound the worst-case cost in Gas. GRuB's decision-making components run on the untrusted cloud off-chain for lower Gas, and employs a security protocol to authenticate the data transferred between the blockchain and cloud. We built a GRuB prototype on Ethereum and supported realfi nancial applications. Using the workloads reconstructed from Ethereum transaction history, we evaluate GRuB's cost and show a Gas saving by 10% ∼ 74%, in comparison with the static baselines.

*✉ Corresponding author.
†Work is done when the author is an undergraduate at Syracuse University.

Middleware'20, DEC 2020, DELFT

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
Doi: 10.1145/3423211.3425696

CCS Concepts: • **Security and privacy → Distributed systems security**; • **Information systems → Remote replication**.

Keywords: Blockchains, data feeds, authenticated data structures, data replication, workload awareness, DeFi.

ACM Reference Format:
Kai Li, Yuzhe Tang, Jiaqi Chen, Zhehu Yuan, Cheng Xu, and Jianliang Xu. 2020. Cost-Effective Data Feeds to Blockchains via Workload-Adaptive Data Replication . In *Middleware'20: ACM/IFIP Middleware, Dec. 7-11, DELFT, The Netherlands.* ACM, New York, NY, USA, 15 pages.

1 Introduction

A smart contract is a user program that runs on a blockchain, such as Ethereum [10] and EOS.IO [5]. It holds the promises to expand the blockchain's functionalities from the basic cryptocurrency payments to broader applications in decentralizedfi nance (DeFi), supply chains, online gaming, et al. Feeding external data onto the blockchain, a.k.a. data feed, is an essential task to enable these blockchain applications. Today, data feeds are widely adopted, notably in DeFi. For instance, stablecoins, a cryptocurrency with stable price that sees an explosion of interest (as in Facebook's Libra [11]) and deployment (as in the popular DAI [23] and Tether [24] tokens on Ethereum) since 2019, require feeding real-world asset prices to the blockchain, for instance the *Ether-price feed* used in DAI [23]. For another instance, to enable asset exchange across different blockchains, say allowing a Bitcoin owner to transact with an Ethereum token owner, it entails a *"side-chain" feed* such as BtcRelay [2, 8, 69] to send the recently found Bitcoin blocks onto Ethereum for verifying Bitcoin deposit. There are many other blockchain applications that have been or can be enabled by data feeds, including decentralized insurance [71], tracing supply-chains [17, 65], healthcare [51], transparency logging [14, 30, 66], trustless information-security [15], et al.

Operating today's data feeds can be an expensive business. Specifically, many real-world data feeds generate an intensive stream of data updates at a high frequency (e.g.,

the price updates in seconds and microseconds). Under these data-intensive streams, data feeds, if improperly designed, could cause a heavy use of blockchain and lead to high monetary cost, known as Gas [68]. The expense burdens not only data-feed operators (e.g., ChainLink and MakerDAO) but also the financial applications running on top of the data feeds (e.g., decentralized exchanges such as AmpleForth and Syntheix [48]), eventually transferring to high fees for end users (e.g., users of decentralized exchanges). It is thus imperative to design cost-effective data feeds for scaling blockchain applications to support real-world data-intensive scenarios.

The goal of this work is to explore how a *dynamic, workload-aware* design of data feed can effectively save Gas. The design goal is motivated by 1) the observation that real-world financial applications exhibit highly dynamic workload patterns, which present opportunities to reduce costs — Intuitively, if one can dynamically adjust the location of the data feeds (w.r.t. the blockchain) according to the current data supply-demand, the Gas cost caused by the repeated use of blockchains could be avoided. See the next two paragraphs for a detailed justification. 2) Furthermore, the design space of a workload-aware approach has not been studied in the existing blockchain-systems research. While there is a large body of research works on reducing blockchain costs, notably the layer-two protocols exemplified by payment channels [16, 35, 52, 56] that aim to place application logic off the blockchain, all existing approaches are based on static data placement. That is, the placement of data and computation w.r.t. blockchains stays fixed once the system starts running, and it does not reflect the constant change in the workloads. The design space of a dynamic, workload-aware approach to optimize smart-contract costs for data feeds is an uncharted territory.

This work presents GRuB, a workload-adaptive data replication framework for cost-effective data feeding. The system model is a data pipeline involving three actors: As illustrated in the left part of Figure 1, an off-chain *data producer (DO)* feeds a stream of data updates to multiple *data-consumer smart contracts (DUs)* on the blockchain. The data flow is coordinated by an intermediary *key-value (KV) store* between the DO and DUs. A conventional design of data feed is to realize the KV store in a smart contract that accepts DO's data updates in transactions and DU's queries in contract internal calls. An alternative design is to statically place the KV store off the blockchain (e.g., the static off-chain feed, TownCrier [71]). By contrast, GRuB is a KV store built on *hybrid* storage media: By default, the data updates are persisted on an off-chain cloud storage provider (SP) such as Amazon S3 [1] and upon DU's queries, are brought to the blockchain, buffered in a smart-contract memory. Optionally, the buffered data can be persisted to the smart-contract storage, as a data replica, to benefit future read queries. GRuB's system model is illustrated by the right part of Figure 1. Note that GRuB's system model considering hybrid third-parties (the trusted blockchain and untrusted cloud) should be differentiated from the existing work considering only untrusted cloud services [57, 63] and one trusted cloud service out of multiple clouds [29, 32].

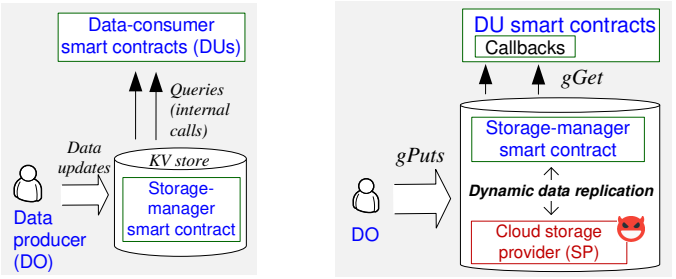


Figure 1. GRuB's system models in comparison with the model of conventional data feed. Green in this figure illustrates smart contracts running on a blockchain, and red is the cloud service provider SP who is the adversary.

The key decision to make in GRuB is whether and when a data record in the feed should be replicated onto the smart-contract storage on a blockchain. *Always* storing a replica of the data being read, on the one hand, can benefit future data reads by avoiding loading data onto the blockchain repeatedly. On the other hand, if there are no future reads, such a data replica would be wasted. Thus, GRuB chooses to replicate data in a *workload-adaptive* manner: If the current workload is dominated by the reads from DUs, the GRuB would decide to store a data replica on the blockchain. Otherwise, if the current workload is dominated by the updates from the DO, the GRuB would decide to avoid replicating data on chain. This design systematically avoids the two most expensive operations in Gas. That is, replicating data on chain under read-intensive workloads can avoid the expensive transactions otherwise needed to bring data onto the blockchain, and evicting data replicas under write-intensive workloads can prevent the expensive storage writes in smart contracts. See Section 2.2 for details on Ethereum's Gas-based cost model and Section 2.3 for a basic measurement study that corroborates our insight here.

Dynamic decision making w.r.t. data replication has been a well-studied research topic in conventional distributed systems. Briefly, a common approach [42] is to model the target system by multiple "sites", and run workload monitoring and decision making distributedly on each site. These solutions lay an important foundation for designing dynamic data-replication in GRuB. However, simply using them as they are in GRuB is insufficient. Notably, existing dynamic replication frameworks are not designed with blockchain's Gas cost model in mind or do not reflect the GRuB's cost to enforce data security (e.g., on untrusted SP off-chain). If used improperly in GRuB, they may lead to excessive costs; for instance, the Gas model charges higher unit cost (e.g., per

word) for "local" operations in smart contract (e.g., on-chain storage updates) than for data movement over the network (by transactions). Such a unique cost characteristic may invalidate the existing design that collocates the decision making with data replicas.

To fill the gap, GRuB presents a Gas-aware data-replication system which places workload monitoring and decision making off the blockchain. We propose a security protocol to guarantee the integrity of workload trace and replication decisions that are transferred from untrusted off-chain SP to the blockchain. The decisions in GRuB are made by a Gas-aware online algorithm that achieves the bounded "Gas competitiveness" – Specifically, the worst-case Gas caused by the data replication following the decision made by this online algorithm is bounded by a small-constant multiplicative factor (e.g., 2) to that caused by an optimal offline algorithm. This work emphasizes building a data-replication *mechanism* supporting sample policies to bound competitiveness. A comprehensive study of policies to configure the mechanism is out of the scope. Overall, GRuB can autonomously run in the hybrid data feeds with changing workloads, while keeping the Gas low.

GRuB's system is generic: To support applications, GRuB exposes an extensible KV store interface (API) that supports Puts from the DO and Gets with callbacks to process queries in a DU contract. GRuB can be built relying on generic interfaces of the underlying systems (similar to an ABI); that is, any blockchain supporting smart contracts and any off-chain storage services supporting KV storage. We have built a GRuB prototype functional with Ethereum [10] and Google LevelDB [13], and used it to enable two popular financial application, namely stablecoin with price feeds and pegged tokens with BtcRelay. Based on the real-world workloads collected from Ethereum, we evaluate GRuB's Gas cost, which shows that GRuB can save up to 67% Gas compared to the static-data-placement baselines. For more extensive evaluation, we build a benchmark by mixing the YCSB workloads. The evaluation under YCSB benchmark shows that compared to the baselines, GRuB can save Gas by 10% $\sim$ 74% depending on specific read-write ratios. GRuB's code is open source.[1]

The contributions of this paper are outline as following:

1. Propose a dynamic, workload-adaptive approach by mixing on-chain and off-chain data storage to optimize the smart-contract costs. To the best of our knowledge, this identifies an unexplored design space in the existing blockchain research.

2. Present GRuB, a Gas-efficient data feed by dynamically replicating data between the hybrid data storage on and off the blockchain. GRuB employs new techniques, a Gas-aware online algorithm for replication decision-making and a security-centric protocol for running the decision components off-chain at a low cost.

3. Validate the applicability of GRuB and evaluate its cost in Gas extensively, by systematically studying real-world applications, building a benchmark suite from real-world traces, and evaluating the costs. The result shows that GRuB can achieve a Gas saving by 10% $\sim$ 74% when compared to static data-placement baselines.

2 Design Motivations

2.1 Preliminary on Motivating Applications

Data feeding enables a blockchain to be able to interoperate with external worlds (i.e., the blockchain interoperability), which further enables a good number of deployed blockchain applications in cross-domain scenarios. We describe two such applications in detail, as an effort to motivate our work.

Stablecoins (on price feeds): Unlike Bitcoin, Ether and other "native" cryptocurrencies, a stablecoin is a cryptocurrency with stable prices. Price stability is the key requirement for real-world adoption of today's cryptocurrencies in realistic applications (e.g., loans, derivatives, and prediction markets). Recently, there is an explosion of stablecoins proposed (e.g., Facebook Libra) and deployed (e.g., DAI [23], Tether [24], and the other 57 stablecoins operational on Ethereum, as of May 2020 [27]).

There are different approaches to realize price stability [36]: A stablecoin can be either directly backed by a stable asset (e.g., USD or gold) or indirectly backed via yet another cryptocurrency. The latter design, named indirectly-backed stablecoin, has the benefit of not relying on a trusted third-party vault off-chain to keep collateral and is adopted in popular stablecoins such as DAI [23] which is indirectly backed by Ether. To manage the price instability of Ether itself, the DAI runs a smart contract on Ethereum that controls the issuance and redemption of DAI. To make each DAI redeemable with one-USD worth of Ether, the DAI smart contract needs to be aware of the current price of Ether (or Ether-USD exchange rate). This is done by a price feed in practice [12], which upload the stream of price updates from a trusted source off-chain, such as Coinbase.[2]

Cross-chain swaps (on side-chain feeds): Supporting asset swaps across multiple blockchains is an important financial application paradigm, enabling asset liquidity on Blockchains. For instance, there are Bitcoin-pegged ERC20 tokens on Ethereum [25] which allow a Bitcoin owner to transact with an asset owner on Ethereum. An efficient approach to enable such applications is the *side-chain paradigm* where blockchain A feeds its produced blocks to smart contracts running on blockchain B. For instance, BtcRelay [2, 8] is such a side-chain feed connecting Bitcoin and Ethereum. BtcRelay style side-chain feeds are widely used in Bitcoin-pegged

[1] https://github.com/syracuse-fullstacksecurity/GRuB

[2] The off-chain party trusted by an indirectly-backed stablecoin performs a much simpler task than that by the directly-backed stablecoin. The former is a price feed, while the latter is a full-fledged vault storing the collateralized asset, subject to the public auditing [36, 53].

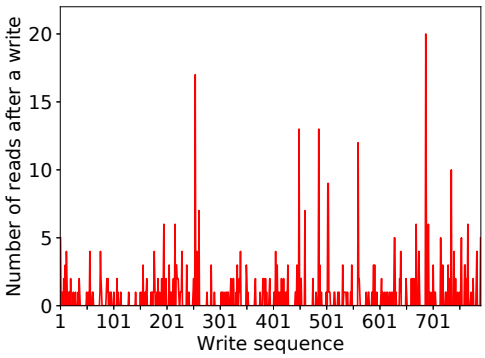


Figure 2. The workloads of ethPriceOracle [18] that feed the MakerDAO stablecoin platform [19] on Ethereum.

ERC20 tokens (e.g., tBTC [4, 22, 28] and others [25, 69]), Ethereum lottery games [3, 9], et al.

Other than the above two classes of data feeds, there are many other uses of data feeds, either deployed or envisioned. For instance, running flight insurances on Ethereum requires data feeds to provide flight cancel/deploy information. Running stock exchanges may require an off-chain order book to feed stock/order prices. In other domains, blockchains are envisioned to support the auditing of transparency logs [14, 66], where the smart contracts running auditing logic need data feeds of log updates from off-chain servers.

Table 1. Distribution of writes by the number of reads followed in the ethPriceOracle trace (#r represents the number of reads per write).

#r	Percentage	#r	Percentage	#r	Percentage
0	70.4%	5	0.76%	10	0.13%
1	16.0%	6	0.63%	12	0.13%
2	6.46%	7	0.25%	13	0.25%
3	2.91%	8	0.13%	17	0.13%
4	1.52%	9	0.25%	20	0.13%

Workloads: In these applications, the workload a data-feed serves consists of data reads from the consumer smart-contracts and the updates from the data producer. One of the motivating observations of this work is that many real-world workloads in data feeds fluctuate widely in the read-write ratio. Here, we present a measurement result as an example. EthPriceOracle [18] is a price feed operational in the Ethereum mainnet and in use to support indirectly-backed stablecoin DAI, as part of the MakerDAO platform [19]. EthPriceOracle allows 14 off-chain accounts to update the price feed and is implemented as a smart contract supporting a price-update function (i.e., `poke()`) and a price-read function (i.e., `peek()`). We collected a call trace of `poke()` and `peek()` between April 25th, 2018 to April 30th, 2018; the collection is done in two means, by running an Ethereum full node and by querying a public Ethereum dataset hosted on Google BigQuery [7]. Figure 2 plots the 5-day trace where each X tick is a data-feed update (i.e., a `poke()` call) and the Y value associated with a X value is the number of data-feed reads (i.e., `peek()` calls) immediately following the write in the call trace. The workload distribution is also summarized in Table 1. It can be seen that the number of reads following a write fluctuate; half of Y values are 0 and 1, but occasionally it also reaches as high as 20 reads after a write.

While this is the case of one particular application, the data-feed workloads being fluctuating commonly apply. Because in a typical data feed, the updates are produced continuously at a regular rate, while the reads from the data consumer smart contract are by demand, which typically come and go in an ad-hoc fashion.

2.2 System Model and Trust Model

In this subsection, we formally describe the system model introduced before. Recall Figure 1 that our system model includes three parties: A data producers (DO), a key-value (KV) store (i.e., the GRuB) and a number of data-consumer smart contracts (DUs). The off-chain DO sends data updates to the KV store, by invoking its function, `gPuts`. A DU smart contract queries the data feed stored in the KV store by issuing a function call to `gGet`. The two functions exposed by the KV store are described by Listing 1.

```
//external call by off-chain DO
bool gPuts(KV[] kvs);
//internal call by smart contract (DU)
KV[] gGet(Key k1, Callback cb);
```

Listing 1. GRuB APIs

Specifically, a single `gPuts` call by the data producer batches multiple KV records in an epoch (e.g., every 1 min.) to update the KV store. A `gGet` call issued by a DU smart contract retrieves KV records by a specified data key and returns its control to an optional callback function in the caller smart contract. The callback function often executes query-processing logic based on the retrieved KV records. Here, note that the caller of `gPuts` is the off-chain data producer and it can be implemented as a remote-procedure call, for instance, in Python. The caller of `gGet` is a smart contract and it can be implemented as a Solidity function.

GRuB is a KV store based on "hybrid" storage media both on and off the blockchain. On the blockchain, it runs a storage-manager smart contract. Off the blockchain, it runs a KV store instance on an untrusted cloud storage provider (SP), such as Amazon S3.

GRuB can be used as a base to support different domain applications. To do so, an application developer writes a DU smart contract encoding the application logic and embedding a query-processor function to be called by `gGet`. GRuB can enable a price feed: Recall Section 2.1 that a price feed supports a price-update function `poke()` and a price-read function `peek()`. These two functions can be mapped to GRuB's `gPuts` and `gGet`, respectively, by modeling the price of each

collateral asset as a KV record (e.g., ⟨Ether, 150USD⟩). Section 4 presents two end-to-end applications built on GRuB.

Trust model: In our system, the primary adversary is the untrusted cloud storage provider who can forge, replay, omit and fork [45] the data sent to the blockchain, in order to break the data integrity. The "data" includes the KV records, proofs and various protocol-specific metadata including collected trace of workloads and replication decisions. We assume high availability among all participating parties and exclude denial-of-service attacks from the scope of this paper. All smart contracts including the application smart contracts and GRuB's storage-manager contracts are trusted in terms of program security (no exploitable security bugs), execution non-stoppability, etc. We also make standard assumption on blockchain security that the blockchain is immutable, fork-consistent and Sybil-secure. The underlying security assumption is that a deployed blockchain system runs among a large number of peers where majority of them are honest peers and compromising the majority is hard.

Table 2. Ethereum's Gas cost w.r.t. different operations [68]: Operations related to data movement (transactions) and storage updates are the most expensive in Gas.

Operation	Gas cost (X is the number of 32-byte words)
Transaction	$C_{tx}(X) = 21000 + 2176X$ $(X < 1000)$
Storage write (insert)	$C_{insert}(X) = 20000X$
Storage write (update)	$C_{update}(X) = 5000X$
Storage read	$C_{read}(X) = 200X$
Hash computation	$C_{hash}(X) = 30 + 6X$

Cost model: The primary cost considered in this work is the cost in using blockchains and executing smart contracts. This paper considers the use of Ethereum. Table 2 presents the Ethereum cost model in Gas (the cost unit in Ethereum). It can be seen the most expensive operations in Gas per word are transactions and storage writes/updates. In our system model, the use of cloud service (SP) may also lead to expenses, which however is much cheaper than that of blockchains: Consider storing one gigabytes in today's cloud storage, which falls under the free tier for all major providers (i.e., Amazon S3, Dropbox, et al), leading to zero-dollar spending, whereas doing the same on Ethereum costs more than $231 million USD (with the Ether price as of Nov. 2019). Because of this, the cloud-service fee in our target application is negligible compared with the Gas cost from blockchains.

Also reducing the Gas of a blockchain application implies improving the throughput of this application, because 1) the transaction throughput of a blockchain is bounded by the total Gas a block can take, such as 10 million gas per Ethereum block; reducing the Gas per operation implies the application can submit more operations in a given time. 2) We assume blockchain is the bottleneck of a target application, which currently takes tens of transactions per second and is much lower than that of conventional computer systems, even for a single machine. Thus, the main goal of this work is to reduce the Gas cost of a blockchain application.

2.3 Motivating Cost Observation

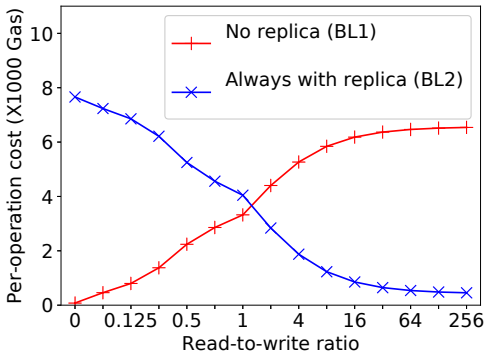


Figure 3. Preliminary Gas measurements of static baselines.

Design Space: This work addresses the design of hybridized data storage over blockchain and SP. We consider the two design baselines: 1) data is only stored on the off-chain SP and is brought into the smart-contract memory when serving `gGet`. This baseline is named BL1. Alternatively, 2) data is stored both on the off-chain SP and on blockchain. The baseline is named BL2. Note that our cost model only considers blockchain-induced cost, Gas, and excludes the off-chain costs including cloud service fee (on SP). Thus, BL2's cost is equivalent to the design of placing data storage only on the blockchain. Note that these two baselines are based on *static* decisions regarding data replication.

Measurement observation: To motivate dynamic data replication of this work, we conduct a rapid measurement study: In this study, we consider the simplest data model involving a single KV record. We implement a simple smart contract on the Ethereum testnet that processes the single KV record with optional on-chain storage. We use an off-chain machine running Ethereum client `geth`, to represent the SP. The two static baselines, BL1 and BL2, are implemented. We use a series of workloads with varying read-write ratios. Each workload is a repeated sequence of $X1$ writes followed by $X2$ reads (all of which are under the single data key). On the one end, we use a write-only sequence , that is, $\frac{X2}{X1} = 0$. On the other end, we use a read-intensive sequence with each write followed by 256 reads $\frac{X2}{X1} = 256$. After driving each workload to our system, we measure the average Gas per operation on BL1 and BL2. We vary the read-to-write ratio ($\frac{X2}{X1}$) and report the measured Gas in Figure 3.

It is clear that as the workload changes from the write-only sequence to read-intensive ones, there is a tradeoff between the two static baselines. When the workload is write-only, BL1 achieves lower Gas per operation than BL2, with cost saving more than 100×. When the workload becomes about every 1.5 read per write (i.e., $\frac{X2}{X1} = 1.5$), the two approaches cost equal Gas. When the workload is more read intensive, such as $\frac{X2}{X1} = 256$, BL2's Gas per operation is $\frac{1}{7}$ of BL1's.

While having a data replica on the blockchain is expected to shift the cost distributions between reads and writes, the striking cost difference it makes (100× and 7×) was surprising to us. This can be attributed to Ethereum's unique cost model: When the workload is write-only, the always-replicate baseline (BL2) incurs expensive operations to update smart-contract storage, which costs $5,000 \sim 20,000$ Gas per word; recall Table 2. When the workload is read intensive, the never-replicate baseline (BL1) incurs expensive transactions to move the latest value of KV record to the blockchain, while BL2 avoids the expense by reading storage data on chain; recall Table 2 that a read from smart-contract storage costs 200 Gas per word while a transaction costs a much higher 2176 per word; let alone the initial cost of 21,000 of an empty transaction.

3 GRuB: System Design and Impl.

GRuB overview: Recall the system model in Section 2.2 that a trusted DO feeds data updates to the GRuB KV store, which is queried by DU smart contracts. The internal system of the GRuB consists of two "planes", as depicted in Figure 4a: 1) A secure-data plane where the DO securely updates the KV store on GRuB by associating data updates with proofs, and a DU smart contract querying the GRuB retrieves query proofs to authenticate (non-replicated) KV records stored on the untrusted cloud provider. The data plane runs a security protocol known as authenticated data structures (ADS; which will be introduced and described in Section 3.3) across the DO, the SP and the blockchain. 2) A control plane which monitors the workloads (data updates and reads), makes replication decisions w.r.t. individual KV records, and stores the decisions as auxiliary states in each KV record, which instructs the data plane to materialize the decisions. The control plane runs on the trusted DO and federates the traces of data reads (from blockchain's native event log recording contract calls) and data updates. GRuB's key component is the online decision-making algorithms running in the control plane.

In this section, we describe the control plane's algorithm design (Section 3.1), the control plane's system design (Section 3.2), the data-plane system design (Section 3.3), overall system properties (Section 3.4) and implementation notes (Section 3.5).

3.1 Online Decision-Making Algorithms

In this subsection, we describe the online decision-making algorithm: Given a sequence of `gPuts` and `gGet` calls, GRuB's decision-making algorithm produces the replication decisions on affected KV records. The replication decision will be actuated as described in the next subsection. The design goal of such algorithms is to reduce the Gas cost of future data reads and writes based on the assumption that the read-/write history will repeat. Intuitively, the algorithm needs to predict the future reads/writes on the KV record, estimate the cost of the two alternative decisions (R or NR) based on the prediction, and pick the one with lower costs as the output. Using the existing online algorithms [42] is insufficient as they are designed without awareness to GRuB's cost in Gas and the cost caused by security proofs. We propose algorithm designs and configurations that are tailored to GRuB's unique costs and that can autonomously achieve bounded worst-case Gas cost. In the following, we present the design and analysis of two algorithms: a "memoryless" online algorithm that resets its state/memory about past reads/writes upon each run, and a "memorizing" online algorithm that remembers the operation history across runs.

Memoryless Algorithm. The memoryless algorithm for replication decision making is described in Algorithm 1. The algorithm internally maintains a list of counters, each for a NR record. The counter counts the number of consecutive reads on the data record that are received since the last write. The algorithm iterates through the read/write trace. Upon a write on a record, say $\langle k, v \rangle$, the algorithm resets the counter of record $\langle k, v \rangle$ back to zero and updates the record's NR. Upon a read on a NR record, it increments its counter. When the counter reaches a preset parameter, K, the algorithm changes the record's state from NR to R and removes the data record from the list of counters.

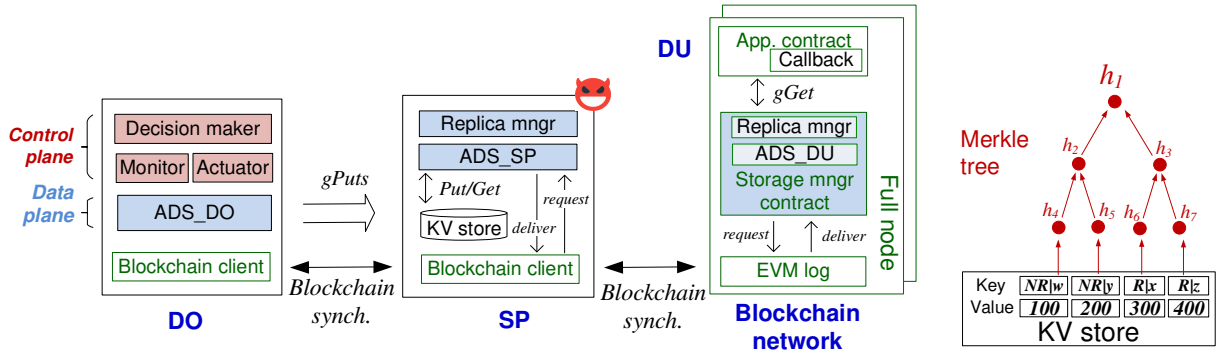
Algorithm 1 MemorylessRepl(*ops*, *count*, *states*)

Input: read/write operations *ops*, read count *count*, and the replication states *states*
Output: updated replication states *states*

```
1:  for all o ∈ ops do
2:      if o.isWrite() then
3:          count[o.key] = 0; states[o.key].set(NR);
4:      else
5:          if count[o.key] < K then
6:              count[o.key] + +;
7:          end if
8:          if count[o.key] ≥ K then
9:              states[o.key].set(R);
10:         else
11:             states[o.key].set(NR);
12:         end if
13:     end if
14: end for
```

Algorithm analysis: To begin with, the competitiveness of online algorithms is the worst-case complexity compared with that of an optimal off-line algorithm. The memoryless algorithm in Algorithm 1 has competitiveness bounded by $1 + K\frac{C_{read\_off}}{C_{update}}$. Here, C_{update} is the Gas to update a byte on the blockchain storage, and C_{read_off} is the unit Gas to send one byte data from off-chain to the blockchain.

Parameter configuration: Parameter K decides the performance of memoryless algorithm. To bound the worst-case Gas, we set K to make the algorithm 2-competitive:



(a) GRuB runs middleware across a smart-contract supported blockchain, SP and DO. **(b)** Merkle tree in ADS_SP.

Figure 4. Overview of GRuB distributed system: Particularly in Figure 4a, the core system components are depicted by shaded boxes in the figure. In blue are data-plane components responsible for data movement and storage, running authenticated data structures (ADS) and managing replicas. In pink are control-plane components that monitor workloads, make replication decisions, and execute the decisions on the data plane.

$$K = C_{update}/C_{read\_off} \quad (1)$$

More formally,

Theorem 3.1. *Memoryless Algorithm 1 with parameters configured by Equation 1 is 2-competitive w.r.t. the Gas cost.*

Due to the space limit, the theorem proof is in Appendix A.

Note that Equation 1 implies a static value for K. In a dynamic replication scheme, using static K, while seemingly counter-intuitive, has the benefit of bounded competitiveness and can also result in actual workload-adaptive cost behavior (as will be evaluated in Section 5 and particularly in Figure 9). There can be other policies to configure K, including setting K dynamic and adaptive to the workload for lower Gas. A comprehensive study of K configuration policies is out of scope of this work, the main goal of which is providing a *mechanism* evaluated by selected policies.

Algorithm 2 MemorizingRepl(*ops*, *rCount*, *wCount*, *states*)

Input: read/write operations *ops*, read counts *rCount*, write counts *wCount* and the replication states *states*
Output: updated replication states *states*

```
for all o ∈ ops do
    if o.isWrite() then wCount[o.key] + +;
    else rCount[o.key] + +;
    end if
    if wCount[o.key] ∗ K′ + D <= rCount[o.key] then
        states[o.key].set(R);
    end if
    if wCount[o] ∗ Y − K′ > rCount[o.key] then
        states[o.key].set(NR);
    end if
end for
```

Memorizing Algorithm. In practice, workloads exhibit temporal locality and can have repeated sequences of read-/write operations. The memoryless algorithm does not capture the temporal locality in the workload by forgetting the past operation history. We propose a memorizing algorithm that exploits the temporal locality in workloads by memorizing the decisions made for similar operations in the past. The memorizing algorithm takes as input the trace of reads and writes. Unlike the memoryless algorithm, the memorizing algorithm needs to take as input the on-chain data reads.

The algorithm, described in Algorithm 2, maintains two counters for each data record, *rCount* and *wCount*. *rCount* (*wCount*) increments when the algorithm, iterating through the read/write trace, encounters a read (write) operation. The algorithm checks two conditions upon each read/write operation: If the condition holds, $wCount*K'+D <= rCount$, the record's state is updated from *NR* to *R*. Here, D is a time window in the past the algorithm looks into to characterize the current workload and to predict the future one. It also resets *wCount* to zero and reduces the value of *rCount* to D. If the condition holds, $wCount * K' - D >= rCount$, the record's state is updated from *R* to *NR*. It also resets *rCount* to zero and reduces the value of *wCount* to D/K'.

Parameter configuration: Similar to the memoryless algorithm, parameter K' is set to the ratio of on-chain write cost to off-chain read cost. $K' = C_{write}/C_{read\_off}$. The other parameter D determines how sensitive the replication state is to the workload. A small D leads to frequent changes of replication state, while a large D leads to a stable setting of replication state.

Theorem 3.2. *Memorizing Algorithm 2 is* $\frac{4D+2}{K'}$*-competitive.*

Due to the space limit, the theorem proof is in technical report [46].

3.2 System Control Plane

The previous subsection describes the online decision-making algorithms and their analysis. This subsection describes how to *execute* the algorithm in the control plane of GRuB. The control plane runs on the DO and is depicted in

Figure 4a. It consists of three essential components: a workload monitor that collects the trace of data reads and writes, the algorithm executor that executes the online algorithm with the monitored trace, and a decision actuator that stores the decisions along with the records in the KV store.

Concretely, the workload monitor running on the DO federates the trace of data updates that occur locally and the trace of data reads from the blockchain history. Here, we consider that the blockchain has a builtin support to log smart-contract invocations, as is the case in Ethereum. The DO runs a blockchain client in full synchronization with other blockchain nodes; the client stores the contract-invocation history, from which the call sequence of `gGet`'es can be accessed. In practice, the DO can run a light blockchain client such as Simplified Payment Verification (SPV) client without downloading the transaction history.

The algorithm output, namely replication decisions, is stored as an auxiliary state in each data record in the KV store. Given a KV record, say $\langle k, v \rangle$, its key is prefixed with an extra bit that indicates whether the record has a replica on the blockchain (i.e., state R) or not (i.e., state NR). The state bit will instruct the data-plane of the system to execute the replication decisions, accordingly (See Section 3.3).

This design assumes a trusted blockchain client whose synchronization with a remote blockchain network is secured by external mechanisms; the client can increase the number of neighbor peers to guarantee the integrity of information synchronized (including blocks and transactions) in the case of compromised blockchain nodes. We dismissed the alternative design by receiving the trace of `gGet` from the untrusted SP which is incentivized to forge the trace and mislead the DO to make a NR decision. Specifically, a NR decision implies more use of the cloud service and the SP can charge higher service fee.

3.3 System Data Plane

This subsection describes the system data plane, in terms of write and read paths. That is, how GRuB handles batched data updates and replication-state transitions from the DO (write path) and data reads from a DU under the current replication states (read path). To guarantee the data authenticity against an adversarial SP, a security protocol, ADS, is adopted in the data plane of GRuB. We begin with a background introduction to ADS.

Preliminary on ADS: An ADS protocol, or authenticated data structure, is a security protocol running among a data owner (ADS_DO), an untrusted service provider (ADS_SP) and multiple data users (ADS_DU). In its most basic form, the ADS_SP accepts data updates (individual KV records) from the ADS_DO and serves exact-match queries (by data keys) issued by ADS_DU. The security properties an ADS guarantee is the authenticity of KV records including record integrity, query completeness and freshness against an adversarial ADS_SP who can forge, replay and omit a query result.

An ADS protocol can be constructed in different ways [44, 50, 50, 54, 55, 62, 72] and GRuB can be easily adapted to these constructions. In our current prototype implementation, we use the common construction based on a Merkle tree. That is, the ADS_SP constructs a Merkle tree on the dataset, each leaf storing the hash of a data record, sorted by their keys. When the ADS_DO wants to update the dataset, she first retrieves the authentication proof of the data key to be updated from the ADS_SP, verifies the data integrity, computes the new leaf hash, and then computes the new root hash based on the proof. The ADS_DO can then safely send the updated data record to the ADS_SP. For data freshness, the ADS_DO can periodically publish the signed root hash to the SP. When a data user, ADS_DU, queries the dataset by a queried key, SP can serve the query by returning the matched KV record and its associated proof. The proof including the latest signed root hash from the trusted ADS_DO can be used to verify the integrity, completeness, and freshness of the query result.

In GRuB, the KV records are sorted by their data keys on SP. Recall that each GRuB record's data key is extended with a prefix of replication state (R or NR). The Merkle tree on ADS_SP is constructed on the key-sorted data layout of records. An example Merkle tree in GRuB is depicted in Figure 4b where the four KV records are first ordered by their NR/R states and then by their actual data keys.

Write path: Given a stream of data updates, DO sends a `gPuts` call every epoch. To prepare the call, DO locally batches the data updates and include them in the single `gPuts` call to be sent by the end of the epoch. Internally, the `gPuts` first notifies the control plane on DO of the latest data updates. Then, for each data update, DO and SP jointly run the ADS protocol to securely update matching KV records.

If all KV records in this batch are in non-replicated state (NR) and there is no update on the replication state, the DO sends only the digest of this batch to call the `update()` function in the storage-manager smart contract. Note that the blockchain node on DO receiving the `update()` call would propagate it to other blockchain nodes. If there are any KV records with replicated state (R), they are included in the `update()` call. If there is any state transition, either from R to NR or from NR to R, such transitions are included in the `update()` call. Receiving the call, the storage-manager contract would insert a new replica to on-chain storage if there is a transition from NR to R. It would evict an existing replica if there is a transition from R to NR.

Read path: Given a `gGet` call from a DU smart contract, all blockchain nodes would execute the storage-manager contract to handle the call. If the requested data key can be matched to a R KV record replicated on the blockchain, the storage manager simply returns the record into the callback function. Otherwise, it emits an event recorded in the

Ethereum log via calling our `request` function. The event can be captured externally by a watchdog service on SP. Specifically, the `request` event is recorded on all Ethereum nodes including the client running on SP. The SP runs an external daemon process (watchdog) that spins on the log to wait for a `request` event. The event triggers the SP to query its local KV store for the requested record before sending it back to the storage-manager contract via calling the `deliver` function. The `deliver` function verifies the integrity of the KV records from off-chain before invoking the callback with the verified KV record.

```
contract GRuB.StorageManager {
  bytes32 rootHash;
  mapping(uint256=>uint256) KVReplicas;
  function gGet(uint256 key, uint256 callback){
    uint256 value = KVReplicas[key];
    if(value != null) callback(key, value);
    //request() emits an EVM log event
    request(key, deliver, callback);}

  function deliver(uint256 key, uint256 value, bool
     replicate, uint256 proof, uint256 callback){
    if(!verify(key,value,proof,rootHash)) return false;
    if(replicate) KVReplicas[key] = value;
    callback(key,value);}

  function update(uint256[] keys, uint256[] values,
     uint256 digest){
    if(msg.sender = DO) rootHash = digest;
    for(int i = 0; i < keys.length; i++){
      if(values[i].replicate) KVReplicas[keys[i]]=values[
     i];
      else delete KVReplicas[keys[i]]; }}}
```

Listing 2. GRuB's storage-manager smart contract.

The pseudo code of storage-manager smart contract is described in Listing 2. The more detailed data-plane workflow is described in the technical report [46].

3.4 Protocol Consistency

In this section, we present the consistency of GRuB protocol and leave more formal proofs to technical report [46].

To describe the protocol consistency, we assume a hypothetical global clock synchronized across the DO and all blockchain nodes. Note that this clock is used as a tool for protocol analysis and is not required in the actual implementation of GRuB.

Blockchain & GRuB model: In a vanilla blockchain, it takes Pt time units to propagate a transaction to all nodes in the blockchain network. It takes an average of B time units to produce a block. Only after F blocks are produced, a transaction is considered finalized in the blockchain network. For instance, in Ethereum, F is 250 and B is $10 \sim 19$ seconds [68].

In GRuB, an epoch E is the time interval in which the DO waits and batches data updates in a transaction.

Consistency between gPut and gGet: Suppose at time t_1 the DO submits a `gPut(k,v)` and at time t_2 a blockchain node Ni executes `gGet(k)`. After $t_2 + P_t + B \cdot F$, assume the execution of `gGet(k)` is finalized on the blockchain.

Particularly, when the record `gGet(k)` accesses is not replicated (NR), time t_2 refers to when the internal call of `gGet(k)` is being entered and returned by the blockchain node (the synchronous execution finishes instantly). When the record `gGet(k)` accesses is not replicated (NR), `gGet(k)` is executed asynchronously and is called back by a `deliver` transaction. In this case t_2 refers to when the `deliver` transaction is executed on node Ni.

Theorem 3.3 (Non-deterministic ordering of concurrent gPut/gGet)**.** `gPut(k,v)` *occurs concurrently with* `gGet(k)`*, if* $t_1 < t_2 < t_1 + E + P_t + B \cdot F$*. With GRuB, the ordering between concurrent* `gPut(k,v)` *and* `gGet(k)` *is non-deterministic and converges to be the same across all blockchain nodes after* $t_2 + P_t + B \cdot F$*.*

Suppose a `gGet(k)` issued by a DU smart contract at local time t on blockchain node Ni returns a set of KV records qs. Query result qs is fresh, w.r.t. delay d, if all KV records matching key k and updated on data owner DO before $t - d$ are included in qs. Here, it assumes a global clock synchronized across the DO and any blockchain nodes Ni. Note that query freshness also implies query completeness here.

Theorem 3.4 (Epoch-bounded query freshness between sequential gPut/gGet)**.** *If* $t_2 > t_1 + E + P_t + B \cdot F$*,* `gPut(k,v)` *is said to occur sequentially after* `gGet(k)`*. Given a* `gGet` *sequentially after a* `gPut`*, GRuB guarantees the* `gGet` *query freshness. Here, the parameters are epoch E, block time B, propagation delay Pt and the number of blocks needed for finality F.*

Supporting delay-sensitive applications: GRuB incurs a maximum delay of E to feed data to the blockchain. Recall that in baseline BL2, data updates are sent directly, without batching, to the blockchain. BL2 guarantees the `gGet` query freshness w.r.t. delay $Pt + F \cdot B$. Applications with the urgent need to feed data can be supported by BL2 where an individual data update is fed to the blockchain immediately after the DO produces it. Note that one can retrofit BL2 to GRuB for supporting these applications where data updates are selected to opt for BL2.

3.5 Implementation Notes

We have built a prototype of GRuB with Ethereum and a Google LevelDB [13] instance. Note that GRuB's design is generally applicable to any storage service exposing a KV store interface (e.g., Amazon S3), an IaaS cloud service allowing user-deployed code (e.g., Amazon EC2) and any blockchains supporting smart contracts. In the prototype, the storage-manager smart contract is implemented in solidity [21]. The off-chain code is written in Python. In particular, the replica manager and ADS protocol relies on a Python binding to interact with the underlying LevelDB [13]. In practice, we use the suggested transaction fee (e.g., 21000 Gas) and Gas price (i.e., 2 GWei) in the evaluation (Section 4 and Section 5), which are sufficient for Ethereum Ropsten

testnet [20] to accept our transactions. How to set Gas price under more adversarial settings such as DoS attacks is out of the scope of this paper.

4 Case Studies

We have built two real applications on GRuB. One is an Ether-backed stablecoin based on a price feed by GRuB and the other is a cross-chain token exchange between Ethereum and Bitcoin based on a BtcRelay style side-chain feed.

4.1 Stablecoins based on Price Feeds

Recall that indirectly-backed stablecoins require feeding the price of the asset that backs the stablecoin. For instance, in stablecoin platform MakerDAO, each currency unit, a DAI, is pegged and redeemable to one-USD worth of Ether. Issuing and redeeming DAI requires Ether price feeds. We build a GRuB-based price feed and use it to support a custom stablecoin SCoin that simulates a simpler DAI.

Specifically, we build a price feed based on GRuB where the KV records store the prices of different assets including Ether. SCoin is implemented as a custom ERC20 token whose supply (in terms of token issuance and redemption) is controlled by a smart contract we build, listed as SCoinIssuer. The smart contract issues SCoins upon receiving Ether payments from an external buyer (i.e., `issue` function), and upon a seller's request to redeem an SCoin, transferring one-USD worth of Ether to the seller before destroying the SCoin (i.e., `redeem` function). To make sure SCoin is pegged and redeemable to one USD, the smart contract needs to read the Ether price at the time of issuance and redemption, as well as requiring over-collateralization and locking up remaining Ether. This implements a minimalist MakerDAO based on the working example in [36].

Cost evaluation: For Gas evaluation, we implemented three price feeds, including GRuB and the two static baselines (BL1 and BL2). We used the call trace of a real price feed, ethPriceOracle [18]. Recall Section 2.1 that this trace records the Ether-price updates and reads from April 25th, 2018 to April 30th, 2018. In our experiment, we set up multiple assets in the price feed: In practice, there are many assets that can be used to back a stablecoin, such as more than 2500 tokens [6] just on Ethereum, fiat currencies (e.g.,USD, Japanese Yen, Euro) and various commodities (gold). We thus set up a KV store of 4096 records in the price feed, each presenting an asset and its price ($\langle asset\_name, price \rangle$). In this setup, a `gPuts` batches price updates of 10 assets, which we use duplicates of the Ether price updates. Each `peek()` call in the trace issues a `gGet` invocation with with a callback to SCoinIssuer's `issue()` or `refund()`, at the equal chance. By this means, we drive the call trace into GRuBPriceFeed and SCoinIssuer.

The result illustrated in Figure 5 shows that GRuB consistently achieves the lowest Gas per operation among the three. Most of the time, BL1 and GRuB achieve lower Gas than BL2. The exception is around epoch 11 when it involves more data reads that put BL1 at disadvantage. Even in this case, GRuB achieves lower Gas than BL2.

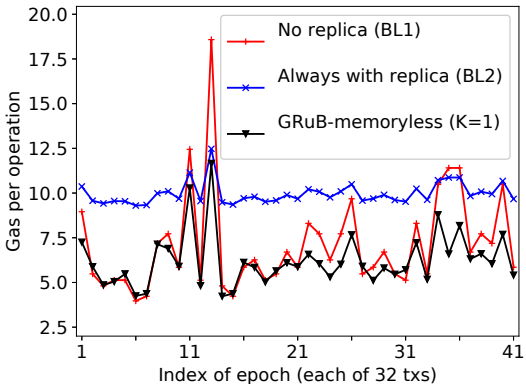


Figure 5. Gas under the 5-day trace (ethPriceOracle).

Table 3. Gas at the data-feed layer and Gas of the end application: M denotes million of Gas.

	Price feed	SCoinIssuer
BL1	$83M$ (+64%)	$86M$ (+67%)
BL2	$55M$ (+11%)	$56M$ (+8.7%)
GRuB	$50.6M$	$51.7M$

Table 3 shows the Gas cost at the data-feed layer and in the end application (SCoinIssuer). It can be seen while SCoinIssuer adds Gas due to application-specific logic, the Gas saving at the data feeding layer is still quite significant.

4.2 BtcRelay Side-chains and Pegged Tokens

BtcRelay feeds Bitcoin blocks to Ethereum and is an important building block for Bitcoin-pegged tokens on Ethereum. We use GRuB to enable BtcRelay by storing the mappings of block hash and Bitcoin block header in the KV store. The DO runs a trusted off-chain Bitcoin client that gets notified every time a Bitcoin block is found.

Based on this data feed, we build a Bitcoin-pegged ERC20 token as an application. The DU smart contract is a simple ERC20 token that supports the operations of mint and burn that consume Bitcoin blocks from the feed: A token-mint (token-burn) operation requires verifying the inclusion of a Bitcoin-deposit (Bitcoin-redeem) transaction against recent Bitcoin blocks from the feed.

Building benchmarks: We collected the trace of transactions to mint/burn eight Bitcoin-pegged tokens known from etherscan.io [26]. The transactions are obtained from Ethereum ETL service on Google BigQuery [7]. We then build a token-contract workload benchmark via joining Bitcoin blocks with Ethereum transactions; the detailed methodology is described in the technical report [46]. The built benchmark contains the read/write sequence to a smart-contract variable storing a stream of Bitcoin block headers.

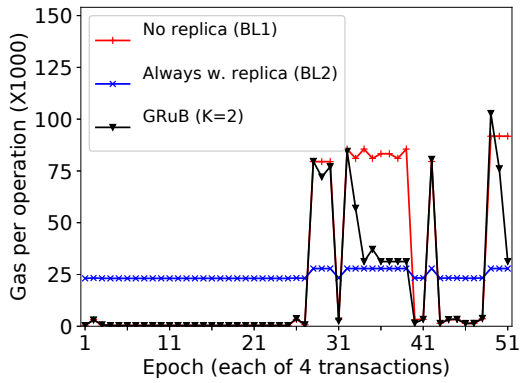


Figure 6. GRuB under the BtcRelay trace.

Experiment results: We measure the Gas cost by GRuB under the workload. We set up an epoch that contains four transactions and drive the established benchmark to our experiments. Particularly, unlike the ethPriceOracle, the BtcRelay workload does not overwrite existing records, but instead appends updates to them. We configure GRuB with reusable storage upon replicating a record. To make the room, previously replicated records unaccessed for a period are evicted.

The result of Gas cost per operation is reported in Figure 6. The trace of the first 25 epochs is write-intensive. In this phase, BL1 outperforms BL2, and GRuB converges to BL1. From epoch 26 to epoch 50, the trace becomes more read-intensive. And BL2 outperforms BL1, and GRuB gradually converges to BL2 (at epoch 34). Overall, GRuB's Gas saving is 56.7%/14.5% compared with BL1/BL2.

5 Cost Evaluation

This section presents the experiments for evaluating the Gas of GRuB. Specifically, our experiments are designed to answer the following questions:

1. How fast will GRuB converge to changing workloads?
2. How sensitive is GRuB's cost to the various parameters that GRuB exposes?

We perform experiments under microbenchmarks (Section 5.1) and macro-benchmarks with YCSB [37] (Section 5.2).

5.1 Microbenchmarks: Converged Gas under Repeating Workloads

In this subsection, we evaluate GRuB's Gas under repeating workloads. We generate the workload that consists of repeated reads and writes under a fixed ratio. Under such workloads, GRuB makes the same decisions, and the Gas becomes converged. Our goal is to evaluate the converged Gas under different factors.

Read-to-write ratio: In this experiment, we evaluate the Gas with different read-to-write ratios. For comparison to GRuB, we consider both baselines of static data replication (i.e., BL1 and BL2). Also, we consider the two baseline designs for dynamic data replication that respectively store on the

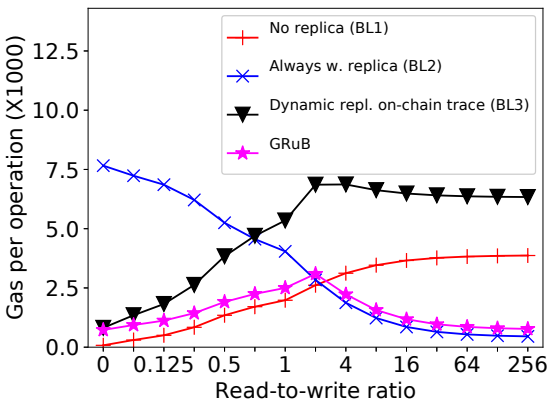


Figure 7. GRuB's Gas with varying read-write ratios.

Blockchain, the trace of reads and writes, and the trace of reads. In each experiment, we drive the synthetic workload of a specific read-to-write ratio to the system and measure the total Gas. We report the average Gas per operation.

In the results reported in Figure 7, baseline BL1 (BL2) has its Gas increased (decreased) as the workload shifts from write-intensive to read-intensive. There is a crossover between BL1 and BL2 when the workload's read-to-write ratio is around 2. GRuB's Gas is slightly higher than BL1 for the read-to-write ratio smaller than 2 and is slightly higher than BL2 for the ratio larger than 2. Note that choosing the one between BL1 and BL2 with lower Gas constitutes an ideal, Gas-optimal dynamic-replication scheme. In this sense, GRuB's (converged) Gas is close to the optimal case. Comparing with the two dynamic-replication baselines, GRuB saves Gas significantly: Especially in read-intensive workloads, GRuB's Gas savings can reach an order of magnitude.

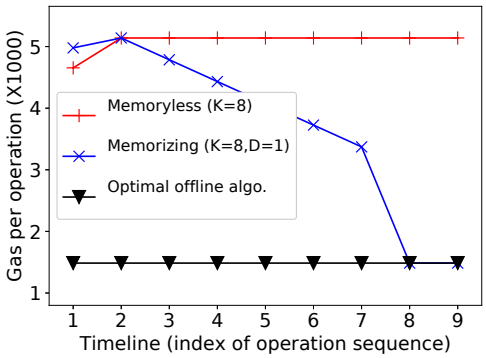


Figure 8. Gas of GRuB under memoryless and memorizing algorithms (with repeating workloads).

Choice of the algorithm: In this experiment, we evaluate how the choice of algorithms affect GRuB's Gas. Recall that we proposed two decision-making algorithms, and they differ in that the memoryless (memorizing) algorithm decides without (with) remembering the historical operations. To contrast the two algorithms to the maximal degree, we use

the following experimental setting: We set parameter $K = K'$ and use the workload of read-to-write ratio $K + 1$. We drive the workload to GRuB with the two different algorithms. Figure 8 reports the Gas per operation along with the timeline (indexed by transactions, each encoding 32 operations). It can be seen GRuB with the memoryless algorithm incurs constant Gas, which is about 5 times higher than the optimal offline decision-making (whose Gas is calculated in a similar way with the previous experiment in Section 5.1). GRuB, with a memorizing algorithm, configured with $K' = 8, D = 1$, initially has a similar level of Gas consumption with memoryless GRuB, and then gradually reduces the Gas close to the optimal algorithm.

5.2 Macro-benchmarks on YCSB

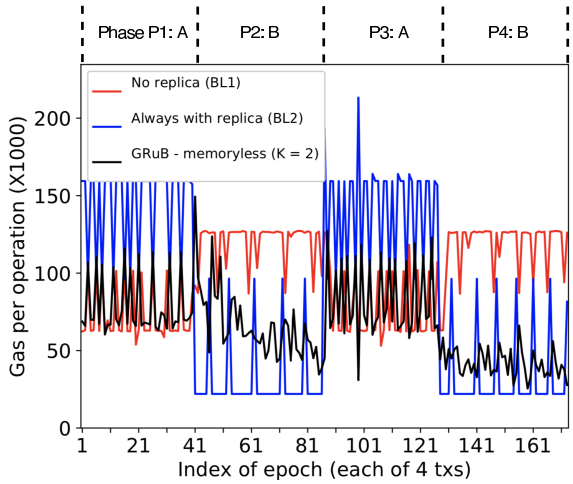


Figure 9. GRuB under mixed YCSB workloads (A and B).

Table 4. Aggregated Gas for mixed YCSB workloads.

Workload	BL1	BL2	GRuB
A, B	1438,130,508 (+31.6%)	1588,684,289 (+45.4%)	1092,576,982
A, E	1400,290,302 (+25.7%)	1936,114,585 (+73.8%)	1114,217,927
A, F	1746,854,231 (+54.1%)	1252,009,322 (+10.4%)	1133,858,720

This set of experiments are designed to evaluate the Gas of GRuB under mixed YCSB workloads. YCSB [37] is an industrial-strength benchmark providing six KV-store workloads, codenamed from A to F, that model some real workloads in Yahoo cloud services. We use YCSB workloads to evaluate GRuB, because of the following reason: GRuB exposes the same KV-store API with most cloud-storage services and brings trustworthiness to these services. Thus, GRuB can and should be a secure alternative to host cloud workloads, especially for security-sensitive applications. In preparing GRuB macro-benchmarks, we mix multiple YCSB workloads, for instance, Workload A and E.

In our experiments, we used three combinations: mixing Workload A and B, mixing Workload A and E, and mixing Workload A and F. Workload A/B/E/F respectively feature 50% reads/95% reads/95% scans/75% reads as well as different key-distribution strategies. In each mixed workload, we pre-load 2^{16} KV records to the GRuB. In Workloads A,B and Workloads A,E, each KV record is set to be 1024-bytes long. In Workload A,F, each KV record is 32-byte long. Each experiment consists of four phases, in each of which one workload generator (i.e., A/B/E/F) is chosen to produce 4096 operations. We report the average Gas per operation for every four transactions (or an epoch).

We report the experiments results by time-series data in Figure 9 and by aggregate results in Table 4. It can be seen that in the first phase P1 (Workload A of 50% reads), the non-replication baseline, BL1, performs better than replication baseline BL2. In Phase P1, GRuB's Gas is close to that of BL1. In the second phase, when the workload switches to Workload B (of 95% reads), the replication baseline BL2 achieves lower Gas than BL1. In Phase P2, GRuB's Gas is lower than BL1 but higher than BL2: Especially at the beginning of P2, GRuB incurs high Gas because of the decision to replicate a KV record being read. Phase P3 is similar to P1. In Phase P4, GRuB's Gas is much lower than P2 because records being read in this phase may already be replicated in Phase P2. The aggregated result, the Gas per operation averaged overall operations, is reported in Table 4, where GRuB saves 32% Gas of BL2 and 46% Gas of BL1. Note that Figure 9 shows a lower rate of saving than Figure 7, because the YCSB workloads tested here are with more restrictive read-write rates than the synthetic workloads used in Figure 7.

6 Related Work

In this section, we describe two research bodies most relevant to our work: cost-effective blockchain applications and workload-aware data replication schemes.

Cost-effective blockchain applications: It is well known that blockchain has limited throughput in handling transactions [38] and incurs high unit cost to execute smart contracts. To reduce the blockchain costs, general approaches are developed by focusing on a permissioned setting [33, 40, 59, 61], or by sharding the blockchain and other layer-one designs [43, 49, 59]. Unfortunately, these new blockchain designs cannot be integrated with an operational blockchain and are known to be difficult to deploy at scale.

A more practical design paradigm, also more relevant to our work, is the layer-two approaches that aim at reducing the use of blockchain in a domain application, without changing the underlying blockchain mechanisms. Notably, payment channels and networks [16, 35, 52] process multiple micro-payments off-chain with issuing two Bitcoin transactions. There are payment networks adopted in practice, such as Lightning Networks in the Bitcoin mainnet [16]. Teechain [47] is a payment network that offloads the detection of participant misbehavior to trusted hardware off-chain, further reducing the involvement of blockchain and improving the application throughput. Similarly, Tesseract [31] employs off-chain trusted hardware to facilitate the payments and exchanges with lower-level involvement of blockchains, to achieve real-timeness and higher throughput. Beyond the

simple application of payments, there are authenticated data structures proposed, such as TPAD [64] and GEM^2 trees [70], to enable the secure handling of database queries off-chain. These layer-two protocols and systems have their off-chain component statically fixed and are not aware of the changing workloads. By contrast, GRuB is the first work that dynamically replicates data onto blockchain.

Testimonium [60] is a Gas-effective blockchain relay (or in our terminology, side-chain feed) which achieves low Gas by lazily validating blocks from a remote blockchain. Our GRuB differs from Testimonium in two senses: First, our focus on data feeding makes GRuB more generally applicable. GRuB supports applications that rely on real-world data feed (e.g., price-feed based stablecoins) that Testimonium cannot support. Second, Testimonium can be thought of as a static data-replication scheme, as it optimistically stores blocks from the remote blockchain without validation. *TownCrier* [71] is a provable-secure data feed service built on off-chain trusted hardware that connects TLS-certified websites to blockchains. The data-feed storage in TownCrier is always off-chain and it does not address the dynamic data replication as in GRuB. *Gasper* [34] is a compiler-based optimization pass that detects Gas-inefficiency anti-pattern in the generated contract bytecode. The cost optimization in Gasper occurs at the syntactic level while GRuB is aware of application semantics and is specific to data feeds.

Workload-aware data replication: In distributed databases, adaptive data replication [41, 42, 67] has been studied: A framework has been proposed by dynamically monitoring the workload and making replication decisions based on the current workload. Many web applications exhibit skewed data-access patterns. MET [39] is a KV store management system that adapts the system configuration and cloud-resource provisioning to the current workload. In designing P2P-based DNS services, Beehive [58] is a proactive data replication scheme that is tailored for Zipf query distribution and achieves the constant look-up cost. GRuB's dynamic replication scheme is inspired by these classic techniques and addresses the technical challenges when combining these classic techniques with blockchains' cost model.

7 Conclusion

This work presents GRuB, a dynamic data replication scheme that achieves low Gas under changing data-access workloads. GRuB runs a Gas-aware, security-centric control framework off the Blockchain. Evaluation shows GRuB saves Gas by up to 70% compared with existing approaches.

Acknowledgments

The authors appreciate anonymous reviewers and the shepherd of this paper, Alysson Bessani. This work was supported by the National Science Foundation under Grant CNS1815814. Jianliang Xu and Cheng Xu were partially supported by Hong Kong RGC Projects 12200819 and 12201018.

A Appendix: Algorithm Analysis

Proof of Theorem 3.1. We first set up the stage by considering an ideal offline algorithm with optimal cost. This offline algorithm can know the entire sequence of reads and writes in advance, and learn the cost-optimal decision. For instance, it can check given a write, if there are more than K consecutive reads that occur after it (before the next write). If so, it can replicate the record at the time of the write, instead of waiting until K reads as in the online algorithm.

For our online algorithm, the worst-case sequence of reads and writes is that every write is followed by exactly K reads. This is the worst-case for our online algorithm because every data replica made by the algorithm is never read, in other words, the cost of replication is totally wasted without saving any cost (of follow-up reads). In this worst case, the cost of our algorithm is $K * C_{read\_off} + C_{update}$. In this case, the cost of the ideal offline algorithm is C_{update}. Thus, the competitiveness of our online algorithm is $1 + K * \frac{C_{read\_off}}{C_{update}}$. Plugging Equation 1 in, we have a competitiveness is bounded by $1 + \frac{C_{update}}{C_{read\_off}} * \frac{C_{read\_off}}{C_{update}}$, which is equal to 2. □

Proof of Theorem 3.2. We use the same offline algorithm as in proving Theorem 3.1. We consider the following sequence of reads/writes for analyzing the worst-case of our memorizing algorithm. The read-write sequence consists of a series of sub-sequences, where the i-th subsequence is of A_i reads and B_i writes. We will set A_i and B_i such that the algorithm will make "wrong" decisions about data replication: It will decide to replicate the data record when it sees A_i reads, and then not to replicate after seeing the next B_i writes. Because each replication decision is followed by writes, the replica is not being read. In other words, the cost of replication is paid without any cost benefit in serving reads by replica. Each no-replication decision is followed by reads, so the follow-up reads are served at the high cost without data replica. In summary, every decision made by the algorithm does not save the cost of serving the following operations, but still incurs replication cost. Hence, this sequence is the worst case of our algorithm.

In the i-th sequence, when the algorithm sees A_i reads, it satisfies the in-equation $(B_1 + B_2 + ... B_i - 1) * K' <= (A_1 + A_2 + ... + A_i) - D$; When it sees B_i writes, it satisfies the in-equation $(B_1 + B_2 + ... + B_i) * K' > (A_1 + A_2 + ... + A_i) + D$; Combining the two in-equations, we conclude that $A_i > 2D$, $B_i >= A_i / K'$. Finally, the general formula for the i-th sequence is: $A_i = D$ when $4i = 1$; $A_i = 2D + 1$ when $i > 1$; $B_i = (2D + 1)/K'$.

The cost of the i-th sequence in our algorithm is $A_i * C_{read\_off} + C_{update} + (B_i - 1) * C_{update}$, and the cost of the ideal offline algorithm is C_{update}; thus the competitiveness of the memorizing algorithm is $A_i * C_{read\_off} / C_{update} + B_i$, since $C_{read\_off} / C_{update}$ equals $1/K'$, the competitiveness is $(4D + 2)/K'$. □

References

[1] Amazon s3: Object storage built to store and retrieve any amount of data from anywhere. https://aws.amazon.com/s3/.

[2] A bridge between the bitcoin blockchain & ethereum smart contracts. http://btcrelay.org/.

[3] Contract address of ethereumlottery.io on etherscan. https://etherscan.io/address/0x302fE87B56330BE266599FAB2A54747299B5aC5B.

[4] Deposit and redeem btc in defi without intermediaries. https://tbtc.network/.

[5] Eos: Blockchain software architecture. https://eos.io/.

[6] Erc20 token list. https://bloxy.info/list$_t$okens/ERC20.

[7] Ethereum blockchain public dataset (hosted by google bigquery). https://console.cloud.google.com/bigquery?project=bigquery-public-data&page=dataset&d=ethereum$_b$lockchain&p=bigquery-public-data&redirect$_f$rom$_c$lassic=true.

[8] Ethereum contract for bitcoin spv. https://github.com/ethereum/btcrelay.

[9] Ethereum lottery - uses bitcoin blocks to pick the winner (via btcrelay). https://www.reddit.com/r/ethereum/comments/4qql6d/ethereum$_l$ottery$_u$ses$_b$itcoin$_b$locks$_t$o$_p$ick$_t$he/.

[10] Ethereum project. https://www.ethereum.org/.

[11] Facebook libra. https://libra.org/en-US/.

[12] Feeds price feed oracles: External reference prices for the maker platform. https://developer.makerdao.com/feeds/.

[13] Google LevelDB, http://code.google.com/p/leveldb/.

[14] Keybase. https://keybase.io/.

[15] Keybase is now writing to the stellar blockchain. https://keybase.io/docs/server$_s$ecurity/merkle$_r$oot$_i$n$_s$tellar$_b$lockchain.

[16] Lightning network, scalable, instant bitcoin/blockchain transactions.

[17] Maersk and IBM Introduce TradeLens Blockchain Shipping Solution, https://ibm.co/37oj56n.

[18] Maker: Medianizer 2 on etherscan. https://etherscan.io/address/0x729d19f657bd0614b4985cf1d82531c67569197b#code.

[19] Makerdao: Digital currency that can be used by anyone, anywhere, anytime.

[20] Ropstern testnet in ethereum. https://github.com/ethereum/go-ethereum/blob/79b11121a7e4beef0d0297894289200b9842c36c/params/bootnodes.go#L34.

[21] Solidity programming language. https://solidity.readthedocs.io/en/develop/.

[22] tbtc (tbtc) token tracker on etherscan. https://etherscan.io/token/0x1bBE271d15Bb64dF0bc6CD28Df9Ff322F2eBD847.

[23] Token: Dai stablecoin. https://etherscan.io/token/0x6b175474e89094c44da98b954eedeac495271d0f?a=0x1e0447b19bb6ecfdae1e4ae1694b0c3659614e4e.

[24] Token: Tether usd. https://etherscan.io/token/0xdac17f958d2ee523a2206206994597c13d831ec7.

[25] Token tracker bitcoin pegged on etherscan. https://etherscan.io/tokens/label/bitcoin-pegged.

[26] Token tracker: Bitcoin pegged on etherscan. https://etherscan.io/tokens/label/bitcoin-pegged.

[27] Token tracker: Stablecoin. https://etherscan.io/tokens/label/stablecoin.

[28] Trustlessly tokenized bitcoin on ethereum. https://github.com/keep-network/tbtc.

[29] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: a case for cloud storage diversity. In J. M. Hellerstein, S. Chaudhuri, and M. Rosenblum, editors, Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010, pages 229–240. ACM, 2010.

[30] M. Ali, J. C. Nelson, R. Shea, and M. J. Freedman. Blockstack: A global naming and storage system secured by blockchains. In A. Gulati and H. Weatherspoon, editors, USENIX ATC 2016, pages 181–194. USENIX Association, 2016.

[31] I. Bentov, Y. Ji, F. Zhang, L. Breidenbach, P. Daian, and A. Juels. Tesseract: Real-time cryptocurrency exchange using trusted hardware. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019, pages 1521–1538, 2019.

[32] A. N. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. Depsky: Dependable and secure storage in a cloud-of-clouds. ACM Trans. Storage, 9(4):12:1–12:33, 2013.

[33] Y. Buchnik and R. Friedman. A generic efficient biased optimizer for consensus protocols. In ICDCN 2020: 21st International Conference on Distributed Computing and Networking, Kolkata, India, January 4-7, 2020, pages 18:1–18:10, 2020.

[34] T. Chen, X. Li, X. Luo, and X. Zhang. Under-optimized smart contracts devour your money. In IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017, pages 442–446, 2017.

[35] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. M. Johnson, A. Juels, A. Miller, and D. Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contract execution. CoRR, abs/1804.05141, 2018.

[36] J. Clark, D. Demirag, and S. Moosavi. Demystifying stablecoins. Queue, 18(1):39–60, Feb. 2020.

[37] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In SoCC, pages 143–154, 2010.

[38] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. E. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer, D. Song, and R. Wattenhofer. On scaling decentralized blockchains - (A position paper). In J. Clark, S. Meiklejohn, P. Y. A. Ryan, D. S. Wallach, M. Brenner, and K. Rohloff, editors, FC 2016 Workshops, volume 9604 of Lecture Notes in Computer Science, pages 106–125. Springer, 2016.

[39] F. Cruz, F. Maia, M. Matos, R. Oliveira, J. Paulo, J. Pereira, and R. Vilaça. Met: workload aware elasticity for nosql. In Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013, pages 183–196, 2013.

[40] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi. Resilientdb: Global scale resilient blockchain fabric. Proc. VLDB Endow., 13(6):868–883, 2020.

[41] Y. Huang, A. P. Sistla, and O. Wolfson. Data replication for mobile computers. In Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994., pages 13–24, 1994.

[42] Y. Huang and O. Wolfson. A competitive dynamic data replication algorithm. In Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria, pages 310–317, 1993.

[43] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In 2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA, pages 583–598, 2018.

[44] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In SIGMOD Conference, pages 121–132, 2006.

[45] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (sundr). In OSDI, pages 121–136, 2004.

[46] K. Li, Y. Tang, J. Chen, Z. Yuan, C. Xu, and J. Xu. Cost-effective data feeds to blockchains via workload-adaptive data replication. http://arxiv.org/abs/1911.04078. CoRR, abs/1911.04078, 2019.

[47] J. Lind, O. Naor, I. Eyal, F. Kelbert, E. G. Sirer, and P. R. Pietzuch. Teechain: a secure payment network with asynchronous blockchain access. In Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019, pages 63–79, 2019.

[48] B. Liu and P. Szalachowski. A first look into defi oracles. CoRR, abs/2005.04377, 2020.

[49] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena. A Secure Sharding Protocol For Open Blockchains. In CCS 2016, pages 17–30, 2016.

[50] C. U. Martel, G. Nuckolls, P. T. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A general model for authenticated data structures. Algorithmica, 39(1):21–41, 2004.

[51] M. Mettler. Blockchain technology in healthcare: The revolution starts here. In 18th IEEE International Conference on e-Health Networking, Applications and Services, Healthcom 2016, Munich, Germany, September 14-16, 2016, pages 1–3. IEEE, 2016.

[52] A. Miller, I. Bentov, R. Kumaresan, and P. McCorry. Sprites: Payment channels that go faster than lightning. CoRR, abs/1702.05812, 2017.

[53] A. Moin, E. G. Sirer, and K. Sekniqi. A classification framework for stablecoin designs, 2019.

[54] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. In Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008, pages 437–448, 2008.

[55] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables based on cryptographic accumulators. Algorithmica, 74(2):664–712, 2016.

[56] J. Poon and T. Dryja. The bitcoin lightning network: Scalable off-chain instant payments. 2016.

[57] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling security in cloud storage slas with cloudproof. In J. Nieh and C. A. Waldspurger, editors, 2011 USENIX Annual Technical Conference, Portland, OR, USA, June 15-17, 2011. USENIX Association, 2011.

[58] V. Ramasubramanian and E. G. Sirer. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In 1st Symposium on Networked Systems Design and Implementation (NSDI 2004), March 29-31, 2004, San Francisco, California, USA, Proceedings, pages 99–112, 2004.

[59] S. Rüsch, K. Bleeke, and R. Kapitza. Bloxy: Providing transparent and generic bft-based ordering services for blockchains. In 38th Symposium on Reliable Distributed Systems, SRDS 2019, Lyon, France, October 1-4, 2019, pages 305–314, 2019.

[60] M. Sigwart, P. Frauenthaler, C. Spanring, and S. Schulte. Decentralized cross-blockchain asset transfers. CoRR, abs/2004.10488, 2020.

[61] J. Sousa, A. Bessani, and M. Vukolic. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018, pages 51–58, 2018.

[62] R. Tamassia. Authenticated data structures. In Algorithms - ESA 2003, 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003, Proceedings, pages 2–5, 2003.

[63] Y. Tang, T. Wang, L. Liu, X. Hu, and J. Jang. Lightweight authentication of freshness in outsourced key-value stores. In C. N. P. Jr., A. Hahn, K. R. B. Butler, and M. Sherr, editors, Proceedings of the 30th ACSAC 2014, New Orleans, LA, USA, pages 176–185. ACM, 2014.

[64] Y. R. Tang, Z. Xing, C. Xu, J. Chen, and J. Xu. Lightweight blockchain logging for data-intensive applications. In A. Zohar, I. Eyal, V. Teague, J. Clark, A. Bracciali, F. Pintore, and M. Sala, editors, Financial Cryptography and Data Security - FC 2018 International Workshops, BITCOIN, VOTING, and WTSC, Nieuwpoort, Curaçao, March 2, 2018, Revised Selected Papers, volume 10958 of Lecture Notes in Computer Science, pages 308–324. Springer, 2018.

[65] F. Tian. An agri-food supply chain traceability system for china based on RFID & blockchain technology. In 2016 13th International Conference on Service Systems and Service Management (ICSSSM). IEEE, jun 2016.

[66] A. Tomescu and S. Devadas. Catena: Efficient Non-equivocation via Bitcoin. In SP 2017, pages 393–409. IEEE Computer Society, 2017.

[67] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. ACM Trans. Database Syst., 22(2):255–314, 1997.

[68] G. Wood. Ethereum: A secure decentralised generalised transaction ledger.

[69] A. Zamyatin, D. Harz, J. Lind, P. Panayiotou, A. Gervais, and W. J. Knottenbelt. XCLAIM: trustless, interoperable, cryptocurrency-backed assets. In IEEE Symposium on SP 2019, pages 193–210, 2019.

[70] C. Zhang, C. Xu, J. Xu, Y. Tang, and B. Choi. Gemˆ2-tree: A gas-efficient structure for authenticated range queries in blockchain. In 35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019, pages 842–853. IEEE, 2019.

[71] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. Town crier: An authenticated data feed for smart contracts. In ACM SIGSAC Conference on CCS, 2016, pages 270–282, 2016.

[72] Y. Zhang, J. Katz, and C. Papamanthou. Integridb: Verifiable SQL for outsourced databases. In I. Ray, N. Li, and C. Kruegel, editors, Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015, pages 1480–1491. ACM, 2015.