WedgeBlock: An Off-Chain Secure Logging Platform for Blockchain Applications

Abhishek Singh
UC Irvine
abhishas@uci.edu

Yinan Zhou
UC Irvine
yinanz17@uci.edu

Sharad Mehrotra
UC Irvine
sharad@ics.uci.edu

Mohammad Sadoghi
UC Davis
msadoghi@ucdavis.edu

Shantanu Sharma
New Jersey Institute of Technology
shantanu.sharma@njit.edu

Faisal Nawab
UC Irvine
nawabf@uci.edu

ABSTRACT

Over the recent years, there has been a growing interest in building blockchain-based decentralized applications (DApps). Developing DApps faces many challenges due to the cost and high-latency of writing to a blockchain smart contract. We propose *WedgeBlock*, a secure data logging infrastructure for DApps. *WedgeBlock*'s design reduces the performance and monetary cost of DApps with its main technical innovation called *lazy-minimum trust* (LMT). LMT combines the following features: (1) off-chain storage component, (2) it lazily writes digests of data—rather than all data—on-chain to minimize costs, and (3) it integrates a trust mechanism to ensure the detection and punishment of malicious acts by the *Offchain Node*. Our experiments show that *WedgeBlock* is up to 1470× faster and 310× cheaper than a baseline solution of writing directly on chain.

KEYWORDS

Blockchain, Off-chain systems, secure logging, smart contracts

1 INTRODUCTION

Blockchain-based[1] decentralized applications (DApps) are applications that operate on blockchain smart contracts. A smart contract is a program that runs on blockchain where the program's logic, requests, and data are recorded and processed on chain. The interest in DApps grew over the recent few years and DApps have amassed hundreds of thousands of users and hundreds of millions of dollars in assets [3]. DApps span many areas including decentralized finance, supply-chain, and gaming.

Developing DApps faces many daunting challenges. The first is in terms of the monetary cost to write to smart contracts. Every request that is sent to the DApp smart contract incurs a monetary fee—called *gas*—that needs to be paid using the corresponding chain's cryptocurrency (this amount fluctuates but as of the writing of this paper, the average fee to process a transaction on Ethereum is around $2.23 [4].) For this reason, it is discouraged for DApps to write large amounts of data to smart contracts.

There has been a number of prior works that explored the problem of performing data logging for DApps [34, 41, 49, 51, 64]. Some of these efforts rely on writing logging data to blockchain smart contracts directly which leads to high monetary cost and performance latency overhead [49]. Alternatively, recent work explored the idea of utilizing an off-chain component to store the logging data while only storing digests of logs on-chain [34, 51]. This approach avoids writing raw logs on-chain which reduces the monetary cost overhead. The log digests written on-chain cost a fraction of the original cost of writing all data on-chain. At the same time, these digests allow clients to verify the authenticity of off-chain data by comparing it with the on-chain digest.

However, these prior approaches suffer from a high latency overhead that is due to the need to write the digests to the blockchain smart contract before the data can be used by other users. This is because before the digest is written to blockchain, users cannot verify the authenticity of the off-chain data—and a malicious node may lie to users by responding with different data logs. The latency overhead is significant as it can be in the order of a few minutes to hours [44]—part of this overhead is the time needed to wait for the request to be included in a blockchain block and the time to wait for the block to be propagated across the network. In addition to their performance latency drawback, prior blockchain-based logging approaches [34, 41, 51] target a single-producer/single-consumer model that limits their use in DApps that require the ability for many users to interact concurrently with the log.

We propose *WedgeBlock*, a data logging platform for DApps that aims to overcome the challenges of existing bockchain-based logging solutions. Most notably, *WedgeBlock* overcomes the high latency overhead by utilizing a concept that we call *Lazy-Minimum Trust* (LMT). In LMT, the off-chain node writes the digest of a log entry E lazily (i.e., asynchronously in the background) on-chain. Before the digest digest(E) is being written to the smart contract, *WedgeBlock* allows the off-chain node to respond to the user requests for E. This is possible by integrating a trust-proof and penalty mechanism. Specifically, the off-chain node when responding with E (that is not yet written on-chain), provides a signed proof to the user that the off-chain node promises to write it on-chain. If the off-chain node lies and writes a different entry E', then the user can use the received signed proof from the off-chain node to impose a punishment on the off-chain node, such as paying a penalty. By setting this punishment to be severe enough to outweigh the gain of lying, the off-chain node would not be incentivized to lie.

WedgeBlock's technical contribution is on the realization of the concept of LMT for secure DApp logging using the following three design principles.

The first principle is to utilize *Offchain Nodes*—machines that are not part of the blockchain network—to perform computations and/or storage on behalf of the smart contract [1, 34, 34, 41, 51]. This enables overcoming the performance and monetary cost overheads of writing on-chain.

The second design principle is *minimum writing on-chain*. Although logs are written in the off-chain component, WedgeBlock

[1]In this paper, we consider permissionless blockchain technologies such as Ethereum as they are the ones used predominately for DApps.

needs to retain the trust properties of the blockchain smart contract. For this reason, WedgeBlock writes a digest of each log entry to blockchain so that clients can verify the authenticity of a log entry by comparing it with the on-chain digest.

The third design principle is *lazy trust*. The goal of lazy trust is to enable a user to trust the response of an off-chain node even before the digest of its request is written to blockchain. To this end, the untrusted off-chain node signs its responses to clients so that clients can prove receiving such response. Then, if it turns out that the off-chain node lied in its response, the user can use the signed response from the off-chain node to invoke a *penalty smart contract* to punish the *Offchain Node*. The assumption we make in the paper is that if the penalty is high enough to outweigh the benefit of acting maliciously, then this mechanism will act as a deterrent against malicious acts.

It is worth noting that the first two principles have similar counterparts in prior work (as described above and as we overview in Sections 7.1and 7.3.) Although similar, the novelty of *WedgeBlock* stems from the combination of the three principles. Combining the three principles lead to unique design challenges and performance gains compared to applying the first two principles in isolation. In terms of design challenges, the off-chain design needs to include an initial setup phase for the punishment and escrow smart contracts as well as a complex payment strategy for the off-chain node's services. These components are not necessary for many existing off-chain nodes. As we show in the design section, applying these additions involves complexities as they need to be performed carefully to avoid security risks. Also, the minimality aspect of on-chain interactions now needs to be extended so that digests are sufficient to prove maliciousness which makes the punishment and payment smart contracts more complex.

WedgeBlock can be integrated into DApps in various ways. In the typical case, we envision that it can be integrated as a library used in the DApp. We also extend the utilization and usage model of *WedgeBlock* to a service model: *DApp-logging-as-a-service*. In this model, a node can act as a service provider to DApps that wish to maintain a DApp log off-chain. To enable this model, we construct a *Payment smart contract* as an automatic payment system between the DApp application owner and the DApp logging service provider. This smart contract enables potential DApp service providers to easily set up a subscription based micropayment channel to the logging service provider to compensate the service provider for its services (i.e., maintaining the log and responding to requests.) We describe the design and implementation of *DApp-logging-as-a-service* in Section 4.5.

In the rest of the paper, we present background information in Section 2. Then, we present WedgeBlock's design (Sections 3 and 4) followed by evaluation results (Section 5 and 6). We discuss related work in Section 7 and conclude in Section 8.

2 BACKGROUND

2.1 Merkle Tree

Merkle trees [39] allow us to prove the integrity of the data stored by a server. Specifically, they have a mechanism to enable a user to detect whether a data item has been modified (or reordered) since it was initially stored. An example of how the Merkle Tree stores data is shown in Figure 1. The Merkle tree stores data on the leaves. The non-leaf nodes of the merkle tree store the concatenated hash values of their child nodes. Each non-leaf node is also computed using a one-way hash function, thus preventing collision attacks. The order of the data in the Merkle

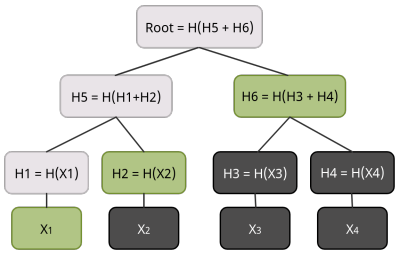

Figure 1: An example of a Merkle Tree.

Tree is captured by the concatenation process while forming the intermediate nodes all the way to the root hash at the top of the tree. Consider node X_1, shown in Figure 1. $H1$ is formed by hashing X_1, and $H2$ is formed by hashing X_2.

Merkle Trees provide the property that if any data in the leaf nodes are changed or reordered, this would result in a completely different root hash. This enables using Merkle Trees to ensure that data that is served from an untrusted server is correct. This is done by returning a merkle proof as part of the server's response. As shown in Figure 1, if a client queries for data X_1, the server responds with the data and nodes shaded in green in the figure. These nodes constitute the merkle proof. The client can use the nodes in the merkle proof to compute the merkle root and validate that the data was not modified in any way by comparing it with the original merkle root that was computed during the construction of the Merkle Tree.

2.2 Blockchain and Smart Contracts

Many public blockchains allow the creation of decentralized programs called *Smart Contracts* [13, 67]. A Smart Contract consists of state (e.g., allocated memory and variables) and functions that can be called to change the Smart Contract state. After being deployed, a Smart Contract is maintained in the blockchain network—i.e., the state and code of the Smart Contract is maintained as part of the ledger. Functions can be called by an external user or by a Smart Contract. Being part of the blockchain ledger, a Smart Contract inherits the tamper-free and immutable nature of blockchain, i.e., any changes made to the smart contract via function calls will be publicly recorded on chain. Typically, a Smart Contract also maintains a balance of cryptocurrency. Another important functionality of Smart Contracts is *emitting events*. Smart contract events can be viewed as a push-based notification system that transmits information from on-chain smart contracts to off-chain subscribers.

2.3 Use Cases

We present two example use cases of *WedgeBlock*.

For the first use case, consider a decentralized IoT solution that aims to create a marketplace for IoT data by connecting IoT publishers with data consumers. This is a growing trend of creating decentralized marketplaces for data [1, 6, 25, 52, 70], some of which focus on IoT data [1, 6, 25]. In these applications, the blockchain application aims to connect IoT data publishers with consumers. *WedgeBlock* helps in such applications by providing an off-chain logging solution that can be used by such applications to store published IoT data for future access by consumers. In particular, IoT publishers send their data to *WedgeBlock Offchain Node*s instead of directly to the blockchain. The *Offchain Node*s establish the authenticity of the logged data by committing digests to the blockchain and providing standardized query APIs for consumers. In many real-life scenarios, the *Offchain Node* is

a third-party service provider that is independent of the publishers and consumers. To support the economical model of such applications, *WedgeBlock* includes an optional Payment smart contract that can be used to manage payments from publishers to *Offchain Node*s.

For the second use case, we look at gaming applications. DApp games use on-chain smart contracts for storing game items as Non-Fungible Tokens (NFTs) which can be bought and sold by players through the game or through online secondary markets. Transactions on these NFTs must be recorded on-chain to keep track of ownership globally. The game itself is hosted on off-chain servers. Due to the high cost of transactions and storage limitations of the blockchain, game data—including user information and game logs—are stored on off-chain servers. Since this data is stored off-chain, the security of this data is dependent on secure logging techniques used by application developers. *WedgeBlock* provides an interface to ensure that these logs are stored securely on *Offchain Node*s while having the verifiability that is offered if data were stored on-chain. An important feature required by DApp gaming applications is correct ordering of events in the log. If two users performed conflicting game actions, the ordering of these events (i.e., which one happened first) is important to maintain. *WedgeBlock* ensures that the order of events that is committed off-chain will be the same one committed on-chain.

3 WEDGEBLOCK OVERVIEW

3.1 System Model and Interface

System Components.

- `Blockchain` and `Smart Contracts`: A public blockchain is used to maintain the state of a tamper-proof smart contract.
- `Offchain Node`: An *Offchain Node* which provides logging services. An *Offchain Node* receives requests to add data to the log and read data from the log.
- `Clients`: We use the term client to include all nodes which use the logging service provided by the *Offchain Node*.

We assume an asynchronous communication model and that all exchanged messages are cryptographically signed.

Programming Interface. WedgeBlock provides two main functions in its interface: (1) Append(in: entry; out: index, proof): this function appends the input entry to the log and returns the index of where it is appended. Entries are batched into log positions. Therefore, the returned index contains information about both the log position (that contains the batch with the entry) as well as the location of the entry inside the batch. In addition to the returned index, the function returns a proof of the append operation. (2) Read(in: index; out: entry, proof): this function returns the entry that corresponds to the input index. Also, it includes a proof of the authenticity of the returned entry.

3.2 Commit Phases

As part of LMT, operations proceed in two phases. The first phase (called *off-chain committed*) denotes when the operation has been received by the off-chain node and a signed response is sent back to the client. At this stage, the entry digest is not yet committed to the blockchain smart contract. Therefore, the only proof returned to the user is a *local proof* that the *Offchain Node* has received the entry and promises to add it to the corresponding index. At this stage, the *Offchain Node* might lie. However, we guarantee that such a lie is going to be detected and a penalty will be imposed. We assume that with a severe enough penalty, this will be a deterrent for malicious acts. The second phase (*blockchain committed*) denotes when the entry's digest is committed to the blockchain smart contract. This means that a proof of being in the *blockchain committed* phase cannot be fabricated by the malicious *Offchain Node*. Once a user receives such proof (called *blockchain proof*), then it is guaranteed to be correct.

Off-chain commitment—before blockchain commitment—provides a weaker notion to the client. This is because it is still possible that the response is fabricated and that a malicious off-chain node would blockchain-commit another entry. However, any such malicious act (i.e., blockchain-committing an entry different than what is off-chain-committed), will result in the client being able to both detect and prove the malicious act. Section 4 shows the details of guaranteeing this property.

3.3 Security and Safety models

In *WedgeBlock*, the *Offchain Node* and clients are not trusted. This means that they can act in any malicious/arbitrary way. This is similar to the byzantine fault-tolerance model [32]. We assume that all communication between clients and *Offchain Node*s is cryptographically signed. We also assume that the integrity of the blockchain network is maintained and that malicious nodes cannot perform attacks to break the immutability and tamper-proof nature of blockchain.

We define the following safety guarantees for WedgeBlock:

Definition 3.1. **Off-chain-commit Safety:** Any off-chain committed response for an index i and entry e pair (called *i-e pair*) from an *Offchain Node* to a client satisfies the following: Either (1) the i-e pair will be the same as the one that will eventually be blockchain committed, otherwise (2) the client (or auditor) can prove that the *Offchain Node* lied and blockchain committed an i-e' pair where $e \neq e'$.

If the *Offchain Node* lied, the proof that the client constructs can be used by the penalty smart contract—that we present later—to withdraw an escrow fund deposited by the *Offchain Node*. Next, is the safety condition for blockchain-committed operations:

Definition 3.2. **Blockchain-committed safety:** Any two clients receiving a blockchain-committed response for an entry with the same index (client 1 receives an i-e pair and client 2 receives an i-e' pair), then it is guaranteed that $e = e'$.

This safety guarantee means that no two clients can disagree about the contents of a log position if they receive a blockchain-committed response.

Security model and attack vectors. we assume a stringent model for malicious behavior which is the byzantine failure model [32], where off-chain nodes can act in arbitrary and malicious ways. We adopt this model as it is a standard model in distributed systems with potentially malicious nodes and utilize it when we discuss the correctness (safety) of the system in Section 4.6 and liveness in Section 4.7. Choosing such a stringent model makes various attack vectors be treated within it—since the byzantine failure model allows arbitrary behavior. We provide examples of this in Sections 4.6 and 4.7.

Punishments. WedgeBlock utilizes the concept of decentralized punishments as a deterrent to malicious activity. An important aspect of introducing a punishment mechanism in DApps, is that they need to be decentralized punishment mechanisms. Having a central authority to enforce punishments would contradict the goal of building a DApp. To this end, WedgeBlock relies on enforcing punishments using smart contracts on blockchain. A node or client that observes a malicious act invokes the punishment smart contract with proof of the malicious act. The smart

contract imposes a penalty—if the proof is correct—by withdrawing from an escrow fund of the malicious node. This requires an initial setup step where the participating nodes setup punishment smart contracts with escrow funds. This mechanism enables a punishment strategy without having to introduce central points of authority. In Section 4.4, we provide more details of the punishment smart contract.

The punishment model that is followed in WedgeBlock is an all-or-nothing (AoN) punishment strategy. In AoN punishments, once a single malicious act is detected, the full punishment amount is invoked, and the contract with the off-chain node is terminated. Because we adopt a lazy detection model—where the time to detection might allow for many malicious acts to be done before the first detection—the configuration of punishment and deposit would be proportional to the window of time where the malicious acts may happen. In WedgeBlock Logging-as-a-Service model, our periodic payment mechanism also acts as a bound on how much time a malicious act can go undetected. The AoN strategy is different than prior punishment mechanisms—such as some layer 2 and Proof-of-Stake algorithms—that rely on 1-to-1 punishment where a punishment corresponds to a single malicious act.

3.4 Design Overview

We now present an overview of the design of *WedgeBlock* (the detailed design is in Section 4.) The core architecture of *WedgeBlock* utilizes off-chain components [8, 15, 23, 40, 68]. In this architecture, the on-chain component represents a blockchain network that processes smart contracts and blockchain transactions. *Offchain Node*s extends the operation of on-chain components by performing compute and/or storage functions.

In *WedgeBlock*, the *Offchain Node* aims to provide data and log storage for clients. Clients send data and log operations to *Offchain Node* and can later access it using read operations. The goal of *WedgeBlock* is to provide the following properties: (1) fast ingestion of data: this aims to overcome the high latency of writing directly on-chain. (2) minimum writing: this aims to reduce the monetary cost of utilizing blockchain by minimizing the amount of data written on chain. Here, we use the term *minimum writing* to refer to only needing to write a digest of arbitrary size data on-chain. As we show in the design, the designer can control the batch size in *WedgeBlock* which leads to controlling the ratio between the size of the digest written on-chain and that data represented by the digest off-chain. Digests are small in size and the batches they represent can be made large to reduce the ratio between the size of on-chain digests and off-chain raw data. (3) trust: this aims to enable utilizing untrusted *Offchain Node*s to achieve the aforementioned goals while ensuring data integrity, hence the data is not tampered with and the safety conditions—defined in Definitions 3.1 and 3.2—are satisfied.

WedgeBlock achieves all the aforementioned properties via **lazy-minimum trust** (LMT), where the *Offchain Node* performs operations locally and provides a proof of the local operation and a promise that a digest of the local operation will be committed to the blockchain smart contract. The purpose of committing the digest to blockchain is to ensure that all clients will agree on the blockchain-committed data. If the *Offchain Node* does not commit the digest of the data to blockchain, then a client can trigger a punishment smart contract using the proof received from the *Offchain Node* that will draw funds from the *Offchain Node*'s escrow account.

As part of initialization, the *Offchain Node* deploys three smart contracts on-chain.

- *Root Record smart contract:* This contract is used to store the digests of the log entries. Each log position can have at most one digest associated with it.
- *Punishment smart contract:* The *Offchain Node* deposits to this contract funds that are withdrawn if a client provides a proof of a malicious act.
- *Payment smart contract:* If a DApp-logging-as-a-service model is deployed, the Payment contract is used to manage payments to the *Offchain Node*.

Processing an operation in WedgeBlock proceeds in two phases that correspond to the two commit phases—off-chain commitment and blockchain commitment. In the first phase, the entry is written locally and a proof is sent back to the client. In the second phase, the *Offchain Node* writes a digest of the entry to the Root smart contract.

Example. consider writing a log entry, e, in log position and index i. In phase 1, the *Offchain Node* writes e locally and responds with a proof to the client. This proof is a signed message that includes the log position and digest of e. Then, the *Offchain Node* sends a request to write the digest of e to the blockchain smart contract in the i^{th} position of a data structure that is maintained by the Root Record smart contract. This write operation can be only performed once for any log position. Therefore, after the digest of e (or the batch containing e if batching is performed) is written to the i^{th} position in the Root Record smart contract, all clients will agree that e is the only blockchain-committed entry in position i. If the *Offchain Node* acted maliciously and blockchain-committed another entry, e', to log position i, then the client can trigger a punishment. This is done by sending the signed proof to a punishment smart contract that matches the proof with the log position in the Root Record contract.

4 *WEDGEBLOCK* DETAILED DESIGN

4.1 Data Model

An append-only log is maintained by the *Offchain Node* to store data. To optimize performance and consolidate operations, the *Offchain Node*processes received append requests in batches. Each log position contains the following fields:

(1) *Log ID*: A monotonically increasing number that uniquely identifies the log position.
(2) *Data List*: Batch of data objects appended by clients.
(3) *Merkle Root (MRoot)*: The merkle root corresponds to the Merkle Tree that is constructed using the Data List.

The append request (A) is defined as $A = (S_p, [n, X])$, where $[n, X]$ represents the payload of the appended data; X is a data object and n is a client-side monotonically increasing sequence number. The sequence number n allows a future reader to specify which data object to read based on the sequence number. S_p is the signature of the client using the tuple $[n, X]$.

Once the *Offchain Node* receives the append request, it generates a stage 1 proof of the log entry comprising of the corresponding *merkle root* and the *Log ID* (denoted as R_{T_i} and i, respectively). Specifically, a response from the *Offchain Node* contains the following tuple (R): $R = (S_e, [X, P_i^X, i])$, where S_e is the signature of the *Offchain Node* on the tuple $[X, P_i^X, i]$; P_i^X is the Stage 1 proof associated with the data object X; and, i is the index where data object X is stored. The client can use the Stage 1 proof and index

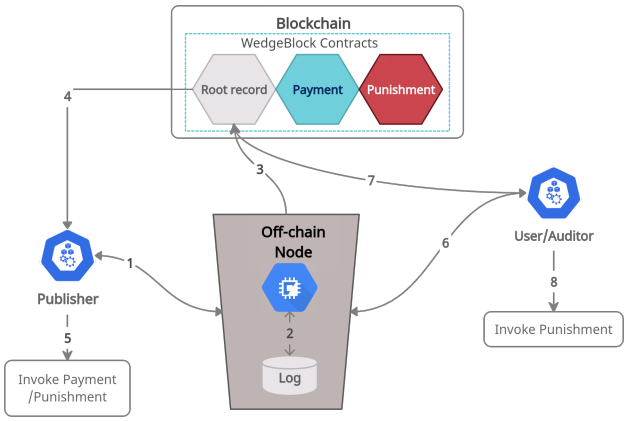

Figure 2: *WedgeBlock* system model and operation flow.

to later verify that the proof matches the digest (MRoot) that is committed to the Root Record contract for index i.

The merkle root R_{T_i} and the index i act as a proof of the integrity of the data in index i and is committed to the blockchain as a stage 2 commit request. The stage 2 commitment record is the tuple V: $V = (i, R_{T_i})$, where the tuple V is committed to the Root Record contract during Stage 2 commitment. A client can query the Root Record contract deployed at the blockchain and verify that the merkle root stored at the blockchain for index i matches the one received as the Stage 1 proof.

4.2 Client Protocols

A Client can perform operations based on three roles. As a Publisher, it sends append operations to the *Offchain Node*. As a User, it sends read operations to the *Offchain Node*. As an Auditor, it sends audit operations, which is a special form of read operations.

Publisher Append Requests. When the publisher wants to append a list of entries, it first transforms the list into append requests (*A*s) and assigns each operation A a unique sequence number n and signs each individual tuple. Then, it sends the list of append requests to the *Offchain Node* to be committed. When the *Offchain Node* responds with a list of responses (*R*s as defined in the Data Model), the publisher verifies each R to ensure its proof and signature are valid. If the publisher receives a valid R for every A, the stage 1 commitment is complete. This procedure is demonstrated in Figure 2 as link #1.

After the first stage, the publisher verifies if the reconstructed merkle root R_{T_i} is consistent with the recorded value in the Root Record contract, shown in Figure 2 as Link #4. Depending on the verification result, the publisher takes one of two actions to complete the second stage of the write request. If the reconstructed root matches the recorded value in the Root Record contract, the request is considered blockchain committed. In the case of any inconsistencies, the publisher invokes the Punishment contract with the conflicting R to claim a compensation from the escrow fund. These actions are shown in Figure 2 as Link #5.

Read Requests. The client can issue a read request of a group of indices together in one operation to the *Offchain Node*. The response from the *Offchain Node* has an identical format to the response for append requests: a list of responses *R*s as defined in the data model section. Figure 2 Link #6 represents this action. The client can then verify that the read items are blockchain-committed by verifying with the Root Record contract.

The Auditor can audit a range of log entries. The auditor sends a signed message specifying the log positions to be audited to the *Offchain Node*. It gets back a response R for every entry stored at the specified log positions. Using these *R*s, the auditor examines if the requested log has been properly maintained by the *Offchain Node*. If the auditor detects any discrepancies between the *Offchain Node* and the Root Record contract—either an invalid merkle tree or a conflicting merkle root—it provides the found evidence to the Punishment contract.

4.3 *Offchain Node* Protocols

Append Requests. All append requests (A) from different publishers are processed in batches. When a publisher sends an append request A (link #1 in Figure 2), the request is buffered as part of a special staging batch called the *current* batch. Once batching is complete, the process of commitment begins. Append requests are processed in two stages as the following.

In stage 1, the new log entry is created for the *current* batch with a new Log ID i. The batch of requests is added to the *Data List* portion of the log entry. The list of requests is then hashed and used to construct a Merkle Tree. Once all the requests have been added to the Merkle Tree, the merkle root R_{T_i} is formed for log entry i. The log entry is then persisted to local storage (link #2 in Figure 2). Then, the *Offchain Node* responds to the clients with the tuple R for each append request A processed in the batch. This is done by computing the Merkle Proof (similar to the example in Figure 1) for each data object X in the *Data List*. This completes the two-way communication shown as link #1 in Figure 2. This completes stage 1 commitment (i.e., off-chain commitment).

For state 1 commitment, the client only needs to ensure that its operations are recorded correctly in the batch that corresponds to R_{T_i}. This is done by checking the Merkle Proof of the client's operation (that contains the leaf node with the operation and the path to the Merkle root.) The client does not need to verify the other operations in the batch. These other operations are either (1) correct and will be validated by their clients, (2) incorrect and will be detected by the clients issuing them, or (3) arbitrary (garbage) data added by the *Offchain Node* that does not correspond to client requests. In the case of arbitrary data, these will not impact non-malicious clients since this is data that is not issued or signed by clients and thus will not be read or used. *Offchain Node*s are not incentivized to include such garbage data as it will increase their costs without being compensated for it.

After off-chain-commit, the stage 2 commitment (i.e., blockchain-commitment) process begins. In stage 2 commitment, the *Offchain Node* takes the Log ID i and the merkle root R_{T_i} to form a tuple (i, R_{T_i}). This tuple is sent to the Root Record contract to be written in the ledger. The process is performed as a smart contract transaction. Once the transaction is committed to the blockchain, stage 2 commitment is complete (link #3 in Figure 2).

Read Requests. There are two types of read requests: The first type is a simple lookup of one or more data objects. For each data object in the request, the *Offchain Node* creates a response tuple R and adds it to the final response message. The second form of the read request is a scan of the log sent by an Auditor. The *Offchain Node* forms a response tuple R for each data object in the requested range and sends it back to the auditor.

4.4 Smart Contract Design

Root Record Smart Contract. The Root Record smart contract serves as an on-chain data store. It has three variables and two methods. The first variable $offchain\_address$ is initialized to a specific Ethereum address—that corresponds to the address of the *Offchain Node*—and is immutable. The second variable $record\_map$ is a hash table that maps log entry digests (i.e.,

MRoots) each to its corresponding unique log position. The third variable $tail\_idx$, initialized to 0, is used to keep track of the most recently updated log index.

The first method $Update - Records$, shown in Algorithm 1, is triggered when the *Offchain Node* sends a request to add a new digest (MRoot) or group of digests. It modifies the $record\_map$ by adding a list of received MRoots and their corresponding log position into the $record\_map$. Line 1 restricts the method to be only callable by the predefined $offchain\_address$. Line 4 checks to ensure that the input $start\_idx$ matches the state variable $tail\_idx$. This is to ensure MRoots are written on-chain sequentially according to the monotonically increasing log indexes. If both checks are passed, the $record\_map$ and $tail\_idx$ are updated accordingly, as shown in the rest of the algorithm.

The second method $Get - Root - At - Index$ is a simple getter function for $record\_map$. It takes an arbitrary key and returns the mapped MRoot value in $record\_map$.

Algorithm 1: The Update-Records Function in the Root Record Contract

Context: This method is invoked by a Ethereum Transaction Txn.
Input: A list $[root_i]$, $i = 0, 1, \cdots, n$, where each $root_i$ is a sequence of bytes.
A starting index $start\_idx$.
Output: None

1 **if** $Txn.sender \neq offchain\_address$ **then**
2 | Method call fails. Returns.
3 **end**
4 **if** $start\_idx \neq tail\_idx$ **then**
5 | Method call fails. Returns.
6 **end**
7 **for** $i \leftarrow 0;\ i < n;\ i \leftarrow i + 1$ **do**
8 | $record\_map[start\_idx + i] \leftarrow root_i$
9 **end**
10 $tail\_idx \leftarrow start\_idx + n$
11 // Method call succeeds. Smart contract state changes.

Punishment Smart Contract. A response R can be incorrect in two ways: first, its merkle root can be different from what was actually stored in the *Offchain Node*'s log. Second, the merkle proof can produce a different merkle root than what is provided. When either one of the two cases occurs, the receiving end can provide the evidence to the Punishment contract to financially punish the *Offchain Node*.

When the smart contract is deployed, it is initialized with the address of the Client node $clientAddress$ and the address of the Root Record smart contract $rootContract$. These addresses are immutable after initialization. The *Offchain Node* deposits ether into the smart contract as an escrow for future punishment. If no punishment occurs throughout the *Offchain Node*'s service, the *Offchain Node* will be refunded the full escrow deposit after the smart contract's termination. Otherwise, the deposit will be transferred to $clientAddress$ as compensation.

The punishment smart contract's core logic is in the $Invoke - Punishment$ method (Algorithm 2). The input parameters for this method correspond to the components of an R response. The method's logic flow is similar to how the Client node verifies R locally, verifying that the merkle root matches the corresponding root in the Root Record contract and verifying that the merkle proof is authentic by verifying that it corresponds to the request's merkle root. When an inconsistency is identified in the input arguments, the smart contract balance is sent to $clientAddress$ as Shown in Lines 7 and 11.

Algorithm 2: Invoke-Punishment

Input: An integer $index$
A byte string $merkleRoot$
A list of byte strings $merkleProof$
A string $rawData$
A bytes string $signature$
Output: None

1 $msgHash \leftarrow hash(index, merkleRoot, merkleProof, rawData)$
2 **if** $recoverSigner(msgHash, signature) \neq Offchain\_Address$ **then**
3 | Punishment fails. Signature is not from the *Offchain Node*. Returns.
4 **end**
5 $recordedRoot \leftarrow rootContract.getRootAtIndex(index)$
6 **if** $recordedRoot \neq merkleRoot$ **then**
| /* Punishment succeeds. */
7 | $clientAddress.call\{value : contract.balance\}$
8 **end**
9 $reconstructedRoot \leftarrow$ reconstruct root using $merkleProof$ and $rawData$
10 **if** $reconstructedRoot \neq merkleRoot$ **then**
| /* Punishment succeeds. */
11 | $clientAddress.call\{value : contract.balance\}$
12 **end**

4.5 DApp-Logging-as-a-Service Model

We present the payment model and implementation of the Payment smart contract to enable a DApp-logging-as-a-service model. In this model, an *Offchain Node* processes the requests in exchange for monetary compensation. The Payment smart contract generates, manages, and settles all the micro-payments needed to be made from the Publisher Client to the *Offchain Node*.

Although blockchain networks like Ethereum allow two parties to make direct payment transactions without needing to do it through a smart contract, such payments are limited in frequency and size due to the high overhead of transaction fees and confirmation delay. Using smart contracts as a medium, existing works on micro-payment channels [20, 21] are able to overcome such limitations. However, micro-payment channels assume the payments to be discrete and require the payer to actively take actions, such as revealing the pre-image of a hashed value to the payee, to unlock the next payment. This assumption do not fit with *WedgeBlock* because *WedgeBlock* needs a micro-payment system that resembles the subscription model where payments are automatically generated continuously until the payer decides to terminate. Therefore, we took a different approach in our payment contract design.

The Payment smart contract maintains the configuration of the subscription payment such as payment frequency and amount. More importantly, the Payment smart contract can acquire timestamps on method invocations from the Ethereum network by reading the blocks' timestamps. These timestamps are used to calculate how much time has elapsed between two method calls. With such information, the Payment smart contract can easily derive the most updated deposit distribution across all involved parties at any given time. When any party attempts to withdraw, the Payment contract guarantees to prevent overdraws. In the remainder of this section, we will explain the relevant variables and methods that enable the Payment contract to achieve this.

The Payment smart contract, at the time of deployment, is initialized with the following variables: $offchain\_address$, $client\_address$, $period$, $payment\_per\_period$ and $max\_overdue\_periods$. The first two variables store the

Ethereum addresses of the two participating parties, the *Offchain Node* and the Client (Publisher) node. (If there are multiple Publishers, they can set up a shared address.) The *period* and *payment_per_period* variables control the payment frequency and amount. The last variable sets a limit on how long the smart contract will tolerate the Client for not paying before considering the Client to be violating the contract. The unit for *period* is seconds while the unit for *payment_per_period* is *wei*—the smallest denomination of ether. For example, if the payer agrees to pay 100 wei per minute and the payee waits at most 120 minutes for overdue payments, the last three variables would be set to 60, 100 and 120, respectively.

After verifying the *Offchain Node* has completed Stage 2 Commitment, the Client node deposits a pre-defined amount of ether into the Payment smart contract and invokes the *startPayment* method to start paying for the service. The *startPayment* method initializes 2 additional variables: *amount_reserved_for_edge* and *payment_start_time*. Variable *amount_reserved_for_edge*, initially 0, is used to indicate what amount of Payment contract balance is withdrawable only by the *Offchain Node*. Variable *payment_start_time* stores the timestamp on when the current stream of micro-payments started.

Once *startPayment* is executed, the Payment smart contract balance is divided into two portions. One is reserved to be withdraw-able only by the $offchain\_address$ while the remaining still belongs to the *client_address*. As time progresses, balance in the second portion virtually moves into the first at a constant rate. However, this process is not done continuously in the background but rather done retrospectively when the Payment contract method *updatePaymentStatus*—shown in Algorithm 3—is invoked. The core logic of this method calculates the correct value for *amount_reserved_for_edge* based on the Payment contract's configuration and current state. Whenever any party tries to withdraw from the smart contract, *updatePaymentStatus* is always invoked first internally to update *amount_reserved_for_edge* which directly determines the withdraw-able amount. When the *Offchain Node* withdraws, the *payment_start_time* is also updated to the timestamp of the block that confirmed the call transaction, essentially resetting the payment calculation.

The *updatePaymentStatus* method also emits important blockchain events under certain conditions. These events are broadcast to subscribed on-chain and off-chain parties so they can react in time. The PaymentStateUpdated event at Line 7 is emitted when there remains enough deposit and it notifies the subscribers about how many more payment periods it can last for. On the other hand, when there are overdue payments, the DepositInsufficient event at Line 10 is emitted to remind the Client node. Lastly, if the accumulated overdue payments exceed the threshold specified by *max_overdue_periods*, the smart contract deems it a violation by the Client node. It will transfer all the remaining balance to $offchain\_address$ before terminating itself with a notification event at Line 14.

Algorithm 3: updatePaymentStatus

Context: The Ethereum transaction that invoked this method is confirmed in block *block* which was mined at $block.timestamp$.
Input: None
Output: None

```
1  elapsedTime ← block.timestamp − paymentStartTime
2  elapsedPeriods ← elapsedTime / period
3  amountReservedForOffchain ←
     paymentPerPeriod ∗ elapsedPeriods
4  if amountReservedForOffchain ≤ contract.balance then
5  |   amountNotReserved ←
   |     contract.balance − amountReservedForOffchain
6  |   remainingPeriods ← amountNotReserved / paymentPerPeriod
7  |   emit event
   |     PaymentStateUpdated(amountReservedForOffchain,
   |     remainingPeriod)
8  else
   |   /* Insufficient deposit                          */
9  |   amountOwe ←
   |     amountReservedForOffchain − contract.balance
10 |   emit event DepositInsufficient(amountOwe)
   |   /* reserve all remaining balance to Offchain Node
   |      */
11 |   amountReservedForOffchain ← contract.balance
12 |   overduePeriods ← amountOwe / paymentPerPeriod
13 |   if overduePeriods ≥ maxOverduePeriods then
   |   |   /* overdue for too long, smart contract
   |   |      automatically terminates                  */
14 |   |   emit event LongOverdue(overduePeriods)
15 |   |   OffchainAddress.call{value :
   |   |     amountReservedForOffchain}
16 |   end
17 end
```

4.6 Correctness

We provide a proof sketch of WedgeBlock satisfying the safety properties in Definitions 3.1 and 3.2.

Theorem 4.1. *WedgeBlock ensures that any offchain-committed request satisfies the safety condition in 3.1.*

Proof. Definition 3.1 states that an offchain-committed request of an i-e pair (where i is the index in the log and e is the entry) must satisfy either one of two properties: (1) the i-e pair is the same as the one that will eventually be blockchain-committed, or (2) the client can prove that an entry e' blockchain-committed at i where $e \neq e'$.

We prove this by contradiction. Assume to the contrary of the two properties above that an offchain-committed request r for an i-e pair is not the same as the one that is eventually blockchain-committed pair, i-e' where $e \neq e'$. Also, contrary to the second property, the client cannot prove that the committed entry e' at i is different than e. This is not possible because the Client can: (1) read the state of the Root Record contract and identify that the blockchain committed entry e' is different from e, and (2) the Client has a signed response R from the offchain node that shows that the offchain node promised to commit the i-e pair. This is a contradiction. □

Theorem 4.2. *WedgeBlock ensures that any blockchain-committed request satisfies the safety condition in Definition 3.2.*

Proof. Definition 3.2 states that any two blockchain-committed requests r_1 and r_2 for the same index i would have the same corresponding entry e. We prove this by contradiction. Assume to the contrary that there are two blockchain-committed requests: r_1 that results in the pair i-e, and r_2 that results in the pair i-e', where $e \neq e'$. Since the two values are different but are in the same log position, this means that the MRoot read from the Root Record contract by r_1, M_1, is different from the one read by r_2, M_2. Because M_1 and M_2 are the MRoots for the same log position and the Root Record contract allows writing to the MRoot of a log position only once with no future updates, then $M_1 = M_2$. This is a contradiction as this indicates that $e = e'$. □

Since the proofs above assume a stringent byzantine failure model that allows any arbitrary behavior by malicious nodes, this

means that various attack vectors cannot threaten the integrity of the application as long as they fall within this definition of arbitrary behavior from utmost f nodes. For example, attacks such as omission, repeating, and truncating attacks would not lead to incorrectness. For example, an off-chain node that truncates the log will be detected by the client that wrote the truncated entries and punishment will be triggered. Due to space limitations, we do not enumerate through the various attack vectors but rely on the generality of the byzantine failure model.

4.7 Liveness

When utilizing untrusted nodes, there are risks in terms of the responsiveness of these nodes. In particular, *omission attacks* denotes a class of attacks where an *Offchain Node* drops or delay responses to requests. Due to the asynchronous communication model we consider, when requests are dropped, it is impossible to determine if the *Offchain Node* is acting maliciously or if the messages were dropped/delayed due to the network. This is a common challenge in systems with malicious (byzantine) nodes. The following are various ways that can be integrated with *WedgeBlock* to reduce the impact of omission attacks:

The first way to tolerate omission attacks is to utilize a number of nodes to act collectively as the *Offchain Node*. For example, a byzantine fault-tolerant (BFT) protocol [17, 32] can be used with a cluster of $3f + 1$ *Offchain Node*s that act collectively to perform the tasks of an *Offchain Node*. This setup would tolerate having up to f malicious nodes that are performing omission attacks by applying existing techniques [18].

The second way is to rely on the economics of omission attacks. In our model, the off-chain node is either run by the owner of the smart contract application or by a third-party contractor that is performing the compute/storage tasks for a monetary compensation. In both cases, there is no incentive for the *Offchain Node* to perform an omission attack. Omission attacks lead to hurting the user experience (studies show that even a small delay would drive users away from an online service [50].) If the *Offchain Node* is operated by the application owner, then omission attacks would lead to driving their customers/users away. Likewise, if the *Offchain Node* is operated by a contractor, fewer users mean less processing and monetary compensation and potentially being replaced with another contractor that is faster.

An extreme case of omission attacks is for an *Offchain Node* to remove/destroy the stored data or blockchain-commit invalid MRoots. The two approaches above can be utilized for this type of attack as well. However, there is an additional strategy that can be utilized in decentralized environments—utilizing decentralized storage solutions [59]. Decentralized storage is a family of protocols that are used to store files across a decentralized network of machines around the world. Decentralized storage can be utilized to maintain a persistent copy of the log that can be accessed even after an extreme omission attack of removing data in the *Offchain Node*.

5 PRACTICAL CONCERNS

Implementation details. We built a prototype to validate *WedgeBlock*'s design. The prototype implements the following design components: the *Offchain Node*, the Publisher node, the User node, the Auditor node and the Smart Contracts. We implement the four off-chain nodes using four Python programs. These programs communicate with each other using the Python gPRC framework [5]. We utilize parallel programming at some sections of the code to improve the prototype's processing speed. The ECDSA signature and verification are applied independently to a large number of data objects so they are executed concurrently using all available CPU cores.

Blockchain network. We deploy *WedgeBlock*'s smart contracts to the Ethereum Ropsten blockchain network. The Ropsten network is one of the most widely used Ethereum test networks that allow researchers and practitioners to test and validate their prototypes. This is a standard approach that is utilized in many academic works [7, 55]. The Ropsten network mining nodes run identical software implementation as those in the Ethereum main network (Mainnet). Executing the same stand-alone smart contract from identical starting states on both Ropsten and Mainnet would produce the same global state transfer and using the same amount of gas. Meanwhile, the Ropsten network circulates free (test) ether coins instead of real ether coins. Therefore we use it to simulate interactions with the Ethereum Mainnet while avoiding paying high fees for prototyping and evaluations.

Availability and traces. Our prototype implementation is available on Github[2]. The details of the *WedgeBlock* smart contracts that we deployed as well as their transaction histories can be viewed publicly on the Etherscan platform for the Root Record smart contract[3], Payment smart contract[4], and Punishment smart contract[5].

Penalty amount configuration. One of the issues needed for a practical deployment of *WedgeBlock* is configuring the amount of escrow funds and penalties during setup. This is an important aspect; however, it is application specific (depending on the workload, participants, off-chain nodes profit, etc) and cannot be treated generally in the context of our current work that focuses on the design of *WedgeBlock*. We defer the question of specifying the amount of penalties to other work that focuses on the economic aspects of on-chain/off-chain interactions. Nonetheless, specifying such amount is orthogonal to the rest of the design and configuration of *WedgeBlock* as long as such configuration of penalties would be a deterrent to malicious acts.

6 EXPERIMENTS AND EVALUATION

6.1 Goals of the experiment

We conduct experiments to evaluate the performance of *WedgeBlock* in terms of performance latency and monetary cost across different system configurations. We also compare the performance of *WedgeBlock* to baseline configurations that mimic the off-chain and on-chain commitment aspects of prior blockchain-based secure logging work [34, 41, 49, 51, 64].

We measure the following metrics. For the smart contracts, we measure the monetary cost of performing stage 2 commitment. For the *Offchain Node*, we measure the throughput of ingesting log entries and the monetary cost. For Clients, we measure the latency of performing append and read operations.

6.2 Experiment Setup

Two Chameleon Cloud machines [30] are used to run the off-chain nodes, one hosting the *Offchain Node* program and the other hosting the three Client programs. Each of the two machines runs on two 64 bit Skylake CPUs with 48 threads each. Each machine provides 192 GiB of RAM and around 300 GB of Storage.

Unless otherwise mentioned, we use the following default parameters. In append experiments, the Publisher node sends 100 batches of append requests to the *Offchain Node*. One batch

[2] https://github.com/alfredd/wedge-block
[3] https://ropsten.etherscan.io/address/0x85bae3688d946d93ed4a31f6e2f60ed74c6035ba
[4] https://ropsten.etherscan.io/address/0x16e07f2fb9d07dca12849f96f295ac1de9e0e681
[5] https://ropsten.etherscan.io/address/0x38d2a28eaf717123c764058759c743bb51547cb0

consists of 10,000 append operations where each append operation contains a key-value pair with a 64-kilobytes key and a 1024-kilobyte value. Each of the 100 batches is measured independently. Then the 100 sets of measurements are averaged to obtain the final measurements for the Publisher. The default batching size utilized by the *Offchain Node* is 2000.

The variables we vary in experiments are the batch size in the *Offchain Node* and the write operation's value size at the Publisher. For each variable, we make multiple runs using different configurations while keeping the other one constant.

6.3 Experiment Result and Analysis

Varying the Batch Size. The first set of experiments measures the performance and monetary cost of *WedgeBlock* while varying the batch size. In this experiment, the size of each append request—formatted as a key-value pair—is fixed to 64B (key) + 1024B (value) = 1088 bytes, approximately 1 KB. We did six runs of experiments, varying the *Offchain Nodes*' batch size between 500 and 10000 append requests per batch.

Figure 3 shows the impact of varying the batch size on the *Offchain Node*'s throughput and monetary cost per operation. In terms of throughput, as we increase the batch size, the throughput decreases. This decrease is within 18% when increasing the batch size from 500 to 10000. The reason for this is that when the batch size increase, the *Offchain Node* produces fewer but larger Merkle Trees. As the Merkle Tree size increases, the time to generate a merkle proof for each write operation increases logarithmically. The red curve shows the throughput of the *Offchain Node* if it first replicates the received write requests to two other machines for a stronger liveness guarantee. Since merkle tree construction takes much longer time than forwarding data batches, the thoughput shows a insignificant decrease when comparing to the *Offchain Node* that only stores data locally.

The right-side plot in Figure 3 measures the monetary cost per operation. As the batch size increases, the cost decreases (increasing the batch size from 500 to 10000 decreases the cost by 87%.) This is because a larger batch size amortizes the cost of many operations into one write to the Root Record contract.

Figure 4 shows the impact of varying the batch size on the Publisher latency. The Publisher sends a batch of append operations to the *Offchain Node*. Then, as append operations are processed, off-chain-commit responses are sent back to the Publisher one by one. The figure shows three lines: (1) *First operation delay*, which represents the latency until the first off-chain-commit response is received, (2) *Last operation delay*, which represents the latency until all off-chain-commit responses are received, and (3) *Stage 1 commitment delay*, which represents the time needed to both receive and process all off-chain-commit responses.

The figure shows that as the batch size increases, the latency increases. This increase is due to the overhead caused by batching at the *Offchain Node*'s side as we described above when discussing the throughput results. The impact of this overhead is different for the first operation compared to the total delay. The increase for the first operation response is higher due to the time needed to generate the Merkle Tree, which is a step that needs to be performed before sending the first response. Our experiments also record the Stage 2 commitment latency at various batch sizes. Since these measurements are mainly determined by the time it takes for the blockchain network to update the smart contract, they do not change significantly across our experiments. The average Stage 2 latency we measure is 43 seconds. The following experiments measure similar results.

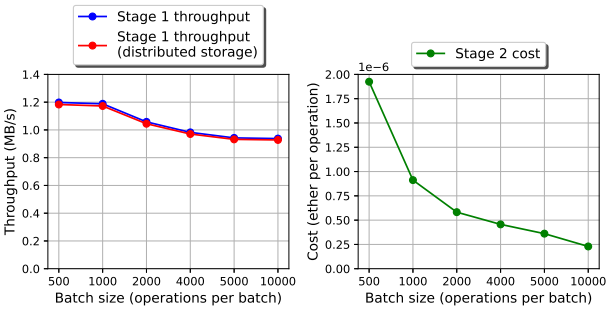

Figure 3: Throughput and cost for various batch sizes

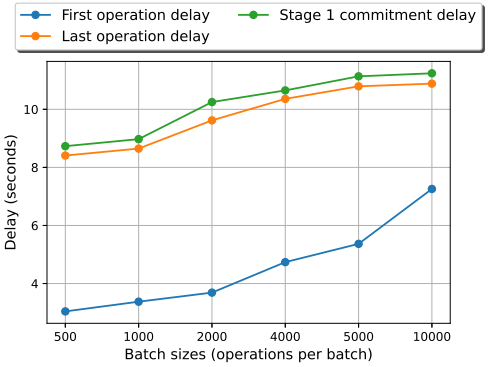

Figure 4: Publisher side latency for various batch sizes

Varying the Value Size. This experiment studies the effect of different write operation's value sizes. In this experiment, the *Offchain Node* always groups incoming operations into batches of 2000 operations. We did four runs of experiments varying the append operations' value size between 512 and 4096 bytes.

Figure 5 shows the impact on off-chain throughput and monetary cost. For throughput (left sub-plot of Figure 5), as the value size increases, the throughput increases as well. This is because as the value size increases, the total workload carries approximately double the amount of bytes to be committed. However, the *Offchain Node* processing is not impacted significantly by the increase in workload; Although larger leaf nodes in a Merkle Tree make the initial hashing step longer to complete, its impact is limited. Once the constant sized hashes are obtained, the remaining time on Merkle Tree construction and proof generation remains the same. This leads to the throughput increase as we increase the value size. We also did these experiments using a *Offchain Node* with a replicated storage mechanism, the red curve in Figure 5 shows that there is insignificant decrease on the throughput. However, the drop in throughput is expected to be larger as value size increases because data transmission time from the *Offchain Node* to the replication nodes increases proportionally with the data size.

In terms of cost per operation (right sub-plot of Figure 5), increasing the value size does not show a significant impact on cost[6]. This is because the size of the append value does not affect the number of Merkle Trees and does not affect the size of merkle roots. Consequently, the *Offchain Node* writes the same amount of data to the Root Record contract in every run. Therefore, we anticipate the total cost to stay relatively constant.

Figure 6 shows the impact of varying the value size on the latency at the Publisher. The figure shows that when the value size increases, the Publisher waits longer to receive all stage 1

[6]Note that the small irregular change in total cost is mostly a reflection of the fluctuation in the Ropsten network's transaction fee.

commitment responses due to the increased communication overhead. However, the verification time is not affected significantly. Stage 1 commitment delay increases by 66% as a result of an 8-time increase of the value size. This shows a trade-off between Publisher latency and *Offchain Node* throughput when the value size changes. In the fourth experiment run with a 4096-byte value size, the *WedgeBlock* runs over 30 minutes without running into any errors, this demonstrates that the *WedgeBlock* is capable of handling large workloads.

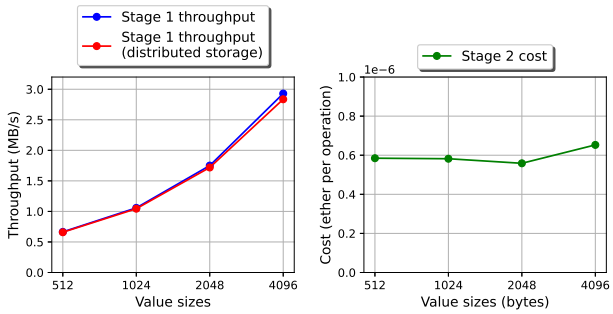

Figure 5: Throughput and cost for various value sizes

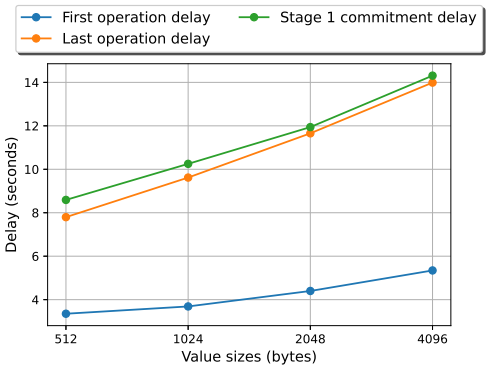

Figure 6: Publisher latency for various value sizes

Varying the request frequency The third experiment studies the *Offchain Node*'s throughput under different workload. In this experiment, the value size and the *Offchain Node*'s batching size are fixed at 1024 bytes and 2000 respectively. In each run, the Publisher node adjusts the number of append operations in each of its 100 requests batches to achieve a certain overall request frequency. For example, if the Publisher node sends 8000 append operations in each batch, the overall request frequency observed by the *Offchain Node* would be 800 requests per second.

Figure 7 shows how the request frequency affects Stage 1 commit throughput. The curve peaks when the frequency is 900 per second, meaning the *Offchain Node* reaches its computational capacity around this point with the given hardware configuration. With a lower frequency, the *Offchain Node* waits in idle in between batches of append operations. When the frequency is larger than 900, the throughput quickly declines due to an accumulation of unprocessed append operations.

Comparison With Prior Approaches. *WedgeBlock* advances the state-of-the-art in secure blockchain-based logging by decreasing the monetary cost overhead with minimum (digest) writing on-chain and by decreasing the latency overhead with lazy trust. Prior blockchain-based approaches can be divided into three categories: (1) on-chain logging (OCL) [49]: these are

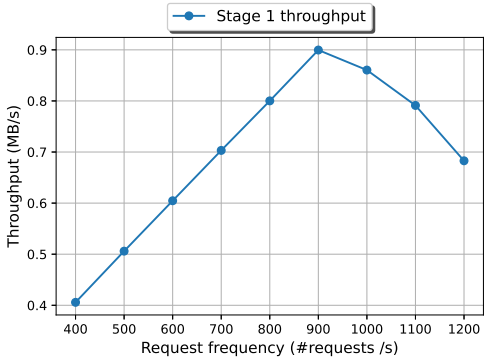

Figure 7: Publisher latency for various request frequencies

approaches that suffer from high latency and monetary cost as raw logs are written on-chain, and (2) synchronous off-chain logging (SOCL) [51]: these are approaches where raw logs are written off-chain and only a digest is written on-chain. However, unlike *WedgeBlock*, these approaches need to wait for the corresponding digest to be written on-chain before reading and trusting the off-chain record. (3) rollup-inspired hybrid logging (RHL) [48]: this approach is inspired from Ethereum Optimistic Rollup and adapted to the problem of logging to allow comparing with *WedgeBlock*. In RHL, the entries are sent to the off-chain node. When the off-chain node responds to the client, this denotes phase 1 commitment (and the latency reported in experiments is this latency). The off-chain node may lie in this response. To detect such malicious act, the client waits for the off-chain node to write the operations and digest on-chain. The client (and any other participant) can use the written entries to verify that the computed digest is correct. If it is not correct, then the client can trigger a challenge smart contract. This challenge smart contract checks whether the digest matches the written operations on-chain. If it does not match, then the challenge is successful. To enable such challenges, there is a challenge period to allow clients to check the accuracy of the digest that lasts for hours to days. This makes the latency of stage commitment much higher than *WedgeBlock*. Also, because the challenge verification relies on having the operations being written on-chain as well, this makes it more costly than *WedgeBlock*.

To compare with these three approaches, we implement the following: for OCL, we implement a smart contract that writes log records on-chain. For SOCL, we implement a client that waits for the digest to be written on-chain to commit. For RHL, we implement a smart contract that writes the operations on-chain. We report two metrics, performance in terms of throughput and monetary cost per operation. The throughput is calculated by dividing the total size of committed data over the time it takes to obtain a successful commitment receipt. For OCL and SOCL, this receipt is the blockchain transaciton confirmation message from the miner nodes. For Wedgeblock and RHL, the Stage 1 commitment proof is used as such receipt as explained in Section 3.

The comparison is shown in Table 1. It shows that *WedgeBlock* improves throughput by up to 1470-times compared to the OCL approach (from 6.6e-4 MB per second to 0.97 MB per second). This is because *WedgeBlock* batch operations, commit after receiving the off-chain response, and write less data on-chain which allows committing more operations. Although the SOCL approach has a relatively higher throughput when compared to OCL, *WedgeBlock* is able to achieve a 5-time increment on the commitment throughput because of using *lazy trust*. Also, the

Value size	Throughput (MB/second)	Cost per operation (ETH)
1024 (OCL)	6.6e-4	0.275
1024 (SOCL)	0.2	1.16e-3
1024 (RHL)	0.97	0.275
1024 (WB)	0.97	1.16e-3
2048 (OCL)	1.4e-3	0.367
2048 (SOCL)	0.46	1.18e-3
2048 (RHL)	1.67	0.367
2048 (WB)	1.67	1.18e-3

Table 1: Commitment throughput and cost of WedgeBlock (WB) compared with prior approaches OCL and SOCL

table shows improvement of SOCL and *WedgeBlock* in terms of monetary cost per operation by up to 310-times compared to OCL (from 0.367 ETH to 1.18e-3 ETH). This is because *WedgeBlock* and SOCL batch operations and only write their digest to the smart contract. On average, when using *WedgeBlock* or SOCL to log operations, each operation only incurs a monetary cost of around one thousandth of an ETH. RHL achieves a throughput that is close to th *WedgeBlock* since it also reports a fast stage 1 commitment latency that does not involve writing to the blockchain. The cost, however, is 310x higher for RHL because it requires writing the operations on-chain.

In terms of latency, OCL and SOCL have a latency that is proportional to the time needed to write on-chain. For RHL, its stage 1 latency is proportional to the time needed to obtain a response from the off-chain node and its stage 2 latency is in the order of hours to days (depending on the design configuration). For *WedgeBlock*, the stage 1 latency—similar to RHL—is proportional to the time to get a response from the off-chain node. However, its stage 2 latency is similar to the latency of SOCL. Although stage 2 latency of *WedgeBlock* is similar to SOCL, because stage 2 is performed asynchronously, *WedgeBlock* can achieve higher throughput. Also, based on our deterrence method that aims to prevent malicious acts from off-chain nodes, clients can rely on stage 1 commitment as a response.

Reading Experiments. We experimented with two read functionalities: random key queries and full log audits. The *Offchain Node* is initialized with 10 million log entries with 64 bytes key and 1024 byte value. The experiment on each functionality consists of six runs in which the *Offchain Node* stores the log entries in batches of different sizes, ranging from 500 to 10000.

For the random key query experiment, the User selects 50000 random entries. We measure the throughput of performing all the read operations which includes communication with the *Offchain Node*, receiving the response, and verifying it. The results are shown in Figure 8. The figure shows that the throughput is between 1800 and 2100 operations per second and that the batch size does not affect the User's read latency significantly.

For the log audit experiment, the Auditor reads and verifies a number of operations. Figure 9 shows the results of reading and verifying operations where the number of operations varies from 10,000 to 200,000 operations. The latency increases linearly with more operations to audit as more work is needed to read and verify operations. The figure also shows how much of the time is spent for verification out of the total latency. In average, 42% of the read time is spent in verification.

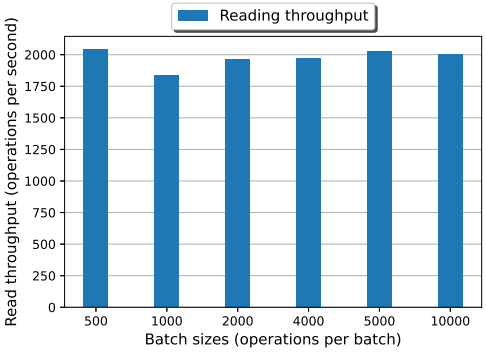

Figure 8: Query latency at various batch sizes

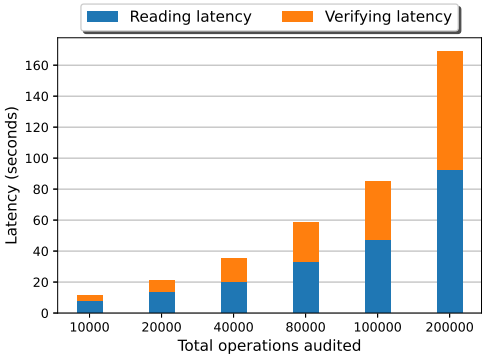

Figure 9: Log audit latency

7 RELATED WORKS

7.1 Blockchain-based Logging

The most related area of work to *WedgeBlock* is the area of blockchain-based logging [9, 34, 35, 41, 49, 51, 64]. Some of these prior works utilize the blockchain itself to write logs [9, 35, 49]. This can lead to high monetary costs and latency overhead when utilizing permissionless blockchains. This lead work following this approach [9, 35, 49] to utilize permissioned blockchain such as HyperLedger [12]. However, permissioned blockchains (i.e., blockchain systems that utilize traditional crash-tolerant or byzantine-tolerant [17, 27, 32] systems) require strong assumptions on the number of failures that can be tolerated which limits the decentralized nature of such protocols and consequently the number of DApps deployed on such networks. An alternative approach is to utilize off-chain nodes to store the raw logs and store a digest of log entries on-chain [34, 41, 64]. hOCBS [41] relies on a permissioned blockchain to overcome the latency and cost overheads—which leads to the challenges we mentioned above about using permissioned blockchains.

PrivacyGuard [64] and Blockchain-enabled Privacy Audit Log (BPAL) [51]—like *WedgeBlock*—use permissionless blockchains in their on-chain/off-chain architecture. To overcome the problem of an untrusted *Offchain Node*, PrivacyGuard [64] uses special hardware (Trusted Execution Environments), while BPAL [51] forces clients to wait until the digest of log entries is written on-chain. These two pieces of work suffer from the limitation of either requiring special hardware [64] or incurring a high latency overhead [51]. *WedgeBlock*, on the other hand, does not require special hardware and allows fast ingestion.

7.2 Secure Logging and Data Processing

Related to *WedgeBlock* is the area of secure tamper-free logging [16, 19, 36, 46, 58, 60]. In [36], for example, the authors

propose an Immutable Forward-Secure Aggregate Authentication (iFssAgg) algorithm. The algorithm builds the log state by encrypting and linking each subsequent state of the log and deleting the intermediate keys. This allows verifying the log by decrypting from the start of the log. One notable application of secure logging has been in cloud-based logging-as-a-service systems, where a cloud service handles the logging functions on behalf of an application [47, 60, 61].

There is a plethora of work in authenticated data and query processing [2, 29, 33, 45, 57, 65, 66, 69]. These methods can be utilized to provide digests and proofs for complex secure computations. This is related to stage 1 commitment in *WedgeBlock*, where such mechanisms can be used to perform more complex processing. This is especially true for such solutions in the context of querying and processing data in hybrid onchain-offchain applications [11, 53, 54].

7.3 Authenticated Off-Chain Protocols

Utilizing off-chain nodes in blockchain has been an area of active work [10, 24, 31, 40, 42]. One challenge that faces utilizing off-chain nodes is that they might be untrusted. This led to a plethora of work in the area of authenticated off-chain protocols[26], where the data and operation that is performed off-chain are augmented with an authenticating strategy. Specifically, this allows off-chain nodes to provide a proof about the authenticity of their data. This includes the use of authenticated and verifiable data structures—such as Merkle Trees [39]—in off-chain nodes [53, 56, 62, 63]. These solutions are orthogonal to *WedgeBlock* and we envision future directions of augmenting these authenticated data structures into the lazy-minimum trust framework of *WedgeBlock*.

Blockchain layer-2 scaling solutions include *rollups* [38, 48] that extend the utilization of authenticated and verifiable data structures in blockchain. One type of rollups is called zero-knowledge rollups (zk-rollups) [14] that utilizes zero-knowledge proofs [22] to enable verifying off-chain computation. However, this entails generating proofs in a complex computation process that makes generating such proofs a lengthy task.

More related to *WedgeBlock* are optimistic rollups that rely on the concept of *fraud proofs*. Optimistic rollups is used as a layer-2 scaling solution of the blockchain itself where blockchain transactions are grouped and committed together while providing a fraud proving mechanism to detect erroneous groupings. Users have a pre-detrmined time period to challenge the outcome of a rollup computation—and if the off-chain node did not perform the correct grouping computation, then users can provide fraud proofs to the blockchain to retract the impact of the rollup. Optimistic rollup, however, target general smart contract and cryptocurrency processing. This generality leads to higher overheads and undesirable characteristics such as: (1) rollup transactions are not considered *committed* (ready) until a challenge period is over to allow users to provide fraud proofs if necessary. This period is suggested to be up to days, which imposes a significant overhead. (2) rollup transactions still need to be written on-chain since the computations they perform are arbitrary and having them on-chain is essential for the fraud proving mechanism. *WedgeBlock*, on the other hand, due to its domain-specific target of data logging, does not suffer from these two challenges; *WedgeBlock* operations are finalized once stage 2 commitment is done (which is much faster than the challenge period), and raw data does not need to be written on-chain for the penalty mechanism (which makes *WedgeBlock* much more cost efficient.)

7.4 Monitoring-Based (Lazy) Trust

Prior work in distributed systems explored the concept of monitoring to detect malicious behavior in the context of byzantine fault-tolerant systems [28, 37, 43]. In these works, the guarantee of detecting malicious acts—with harsh enough punishments—is used as a deterrent to malicious nodes. These works are specific to distributed environments such as edge-cloud systems [43] and distributed consensus [28, 37] which makes them unapplicable to the decentralized blockchain environment we consider. Our work extends these works by making the concept of moniroting-based trust (lazy trust) applicable to decentralized blockchain environments in two ways: (1) it augments the concept of lazy trust with blockchain, where the blockchain smart contract acts as the trusted entity. This entails tackling challenges in performing lazy trust with smart contracts as well as leverage opportunities of monetary punishments through escrow funds. (2) it considers the issue of minimum trust that is essential for the efficient utilization of blockchain as a trusted entity. Without minimizing the amount of data to be sent to the blockchain smart contract, the cost of lazy trust would still be prohibitive.

8 CONCLUSION

We have presented *WedgeBlock*: a blockchain-enabled secure logging system. *WedgeBlock* creates a platform for users to store and retrieve data from untrusted servers. It guarantees that that data corruption can be easily identified and malicious servers punished. *WedgeBlock* also ensures that blockchain usage is cheap and affordable by storing minimal amount of metadata on the blockchain and minimizing the number of blockchain transactions while continuing to provide secure logging services. *WedgeBlock* is intended to become a secure logging component that can be easily adopted by applications. Through our evaluations we show that *WedgeBlock* provides high performance logging service for storage, retrieval and verification of data.

9 ACKNOWLEDGMENTS

This research is supported in part by the NSF under grant CNS-1815212 and a gift from Facebook. We would also like to thank Professor Nalini Venkatasubramanian from the Information Systems Group at UC, Irvine, for her feedback and support for the WedgeBlock project.

REFERENCES

[1] Anylog: Technology. https://anylog.co/technology/.

[2] Aws amazon quantum ledger database (qldb). https://aws.amazon.com/qldb.

[3] Dapp radar rankings. https://dappradar.com/rankings.

[4] Ethereum charts and statistics. https://etherscan.io/charts.

[5] Introduction to grpc. https://grpc.io/docs/what-is-grpc/introduction/.

[6] D. Abadi, O. Arden, F. Nawab, and M. Shadmon. Anylog: a grand unification of the internet of things. In *Conference on Innovative Data Systems Research (CIDR '20)*, 2020.

[7] Z. Abou El Houda, A. S. Hafid, and L. Khoukhi. Blockchain-based reverse auction for v2v charging in smart grid environment. In *ICC 2021-IEEE International Conference on Communications*, pages 1–6. IEEE, 2021.

[8] J. Adler, R. Berryhill, A. Veneris, Z. Poulos, N. Veira, and A. Kastania. Astraea: A decentralized blockchain oracle. In *2018 IEEE international conference on internet of things (IThings) and IEEE green computing and communications (GreenCom) and IEEE cyber, physical and social computing (CPSCom) and IEEE smart data (SmartData)*, pages 1145–1152. IEEE, 2018.

[9] A. Ahmad, M. Saad, M. Bassiouni, and A. Mohaisen. Towards blockchain-driven, secure and transparent audit logs. In *Proceedings of the 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, pages 443–448, 2018.

[10] H. Al-Breiki, M. H. U. Rehman, K. Salah, and D. Svetinovic. Trustworthy blockchain oracles: review, comparison, and open research challenges. *IEEE Access*, 8:85675–85685, 2020.

[11] L. Allen, P. Antonopoulos, A. Arasu, J. Gehrke, J. Hammer, J. Hunter, R. Kaushik, D. Kossmann, J. Lee, R. Ramamurthy, et al. Veritas: Shared verifiable databases and tables in the cloud. In *9th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2019.

[12] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 30:1–30:15, New York, NY, USA, 2018. ACM.

[13] A. M. Antonopoulos and G. Wood. *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.

[14] D. Augot, S. Bordage, Y. El Housni, G. Fedak, and A. Simonet. Zero-knowledge: trust and privacy on an industrial scale. 2022.

[15] R. Belchior, A. Vasconcelos, S. Guerreiro, and M. Correia. A survey on blockchain interoperability: Past, present, and future trends. *ACM Computing Surveys (CSUR)*, 54(8):1–41, 2021.

[16] M. Bellare and B. Yee. Forward-security in private-key cryptography. In *Cryptographers' Track at the RSA Conference*, pages 1–18. Springer, 2003.

[17] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[18] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 277–290, New York, NY, USA, 2009. Association for Computing Machinery.

[19] S. A. Crosby and D. S. Wallach. Efficient data structures for tamper-evident logging. In *USENIX Security Symposium*, pages 317–334, 2009.

[20] M. Elsheikh, J. Clark, and A. Youssef. *Short Paper: Deploying PayWord on Ethereum*, pages 82–90. 03 2020.

[21] H. S. Galal, M. ElSheikh, and A. M. Youssef. An efficient micropayment channel on ethereum. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, page 211–218, Berlin, Heidelberg. Springer-Verlag.

[22] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 203–225. 2019.

[23] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais. SoK: Off The Chain Transactions. Technical Report 360, 2019.

[24] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais. Sok: Off the chain transactions. *IACR Cryptol. ePrint Arch.*, 2019:360, 2019.

[25] P. Gupta, S. S. Kanhere, and R. Jurdak. A decentralized iot data marketplace. *CoRR*, abs/1906.01799, 2019.

[26] S. Gupta, J. Hellings, and M. Sadoghi. *Fault-Tolerant Distributed Transactions on Blockchain*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2021.

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

[28] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: practical accountability for distributed systems. *ACM SIGOPS Operating Systems Review*, 41(6):175–188, Oct. 2007.

[29] R. Jain and S. Prabhakar. Trustworthy data from untrusted databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 529–540. IEEE, 2013.

[30] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[31] R. Kumar, N. Marchang, and R. Tripathi. Distributed off-chain storage of patient diagnostic reports in healthcare system using ipfs and blockchain. In *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*, pages 1–5. IEEE, 2020.

[32] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[33] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 121–132, 2006.

[34] J. C. López-Pimentel, O. Rojas, and R. Monroy. Blockchain and off-chain: A solution for audit issues in supply chain systems. In *2020 IEEE International Conference on Blockchain (Blockchain)*, pages 126–133. IEEE, 2020.

[35] N. Lu, Y. Zhang, W. Shi, S. Kumari, and K.-K. R. Choo. A secure and scalable data integrity auditing scheme based on hyperledger fabric. *Computers & Security*, 92:101741, 2020.

[36] D. Ma and G. Tsudik. A new approach to secure logging. *ACM Trans. Storage*, 5(1), mar 2009.

[37] S. Maiyya, D. H. B. Cho, D. Agrawal, and A. El Abbadi. Fides: managing data on untrusted infrastructure. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 344–354. IEEE, 2020.

[38] O. Marukhnenko and G. Khalimov. The overview of decentralized systems scaling methods. *COMPUTER AND INFORMATION SYSTEMS AND TECHNOLOGIES*, 2021.

[39] R. C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In C. Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, Lecture Notes in Computer Science, pages 369–378. Springer Berlin Heidelberg, 1988.

[40] A. Miller, I. Bentov, S. Bakshi, R. Kumaresan, and P. McCorry. Sprites and state channels: Payment networks that go faster than lightning. In *International Conference on Financial Cryptography and Data Security*, pages 508–526. Springer, 2019.

[41] K. Miyachi and T. K. Mackey. Hocbs: A privacy-preserving blockchain framework for healthcare data leveraging an on-chain and off-chain system design. *Inf. Process. Manage.*, 58(3), may 2021.

[42] R. Mühlberger, S. Bachhofner, E. Castelló Ferrer, C. D. Ciccio, I. Weber, M. Wöhrer, and U. Zdun. Foundational oracle patterns: Connecting blockchain to the off-chain world. In *International Conference on Business Process Management*, pages 35–51. Springer, 2020.

[43] F. Nawab. Wedgechain: A trusted edge-cloud store with asynchronous (lazy) trust. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 408–419. IEEE, 2021.

[44] M. Pacheco, G. A. Oliva, G. K. Rajbahadur, and A. E. Hassan. Is my transaction done yet? an empirical study of transaction processing times in the ethereum blockchain platform. *ACM Transactions on Software Engineering and Methodology*, 2022.

[45] D. Papadopoulos, S. Papadopoulos, and N. Triandopoulos. Taking authenticated range queries to arbitrary dimensions. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 819–830, 2014.

[46] I. Ray, K. Belyaev, M. Strizhov, D. Mulamba, and M. Rajaram. Secure logging as a service—delegating log management to the cloud. *IEEE Systems Journal*, 7(2):323–334, 2013.

[47] I. Ray, K. Belyaev, M. Strizhov, D. Mulamba, and M. Rajaram. Secure logging as a service—delegating log management to the cloud. *IEEE systems journal*, 7(2):323–334, 2013.

[48] T. Schaffner. Scaling public blockchains. *A comprehensive analysis of optimistic and zero-knowledge rollups. University of Basel*, 2021.

[49] L. Shekhtman and E. Waisbard. Engravechain: A blockchain-based tamper-proof distributed log system. *Future Internet*, 13(6), 2021.

[50] W. Stadnik and Z. Nowak. The impact of web pages' load time on the conversion rate of an e-commerce platform. In L. Borzemski, J. Świątek, and Z. Wilimowska, editors, *Information Systems Architecture and Technology: Proceedings of 38th International Conference on Information Systems Architecture and Technology – ISAT 2017*, pages 336–345, Cham, 2018. Springer International Publishing.

[51] A. Sutton and R. Samavi. Blockchain enabled privacy audit logs. In C. d'Amato, M. Fernandez, V. Tamma, F. Lecue, P. Cudré-Mauroux, J. Sequeda, C. Lange, and J. Heflin, editors, *The Semantic Web – ISWC 2017*, pages 645–660, Cham, 2017. Springer International Publishing.

[52] M. Travizano, C. Sarraute, G. Ajzenman, and M. Minnoni. Wibson: A decentralized data marketplace. *CoRR*, abs/1812.09966, 2018.

[53] H. Wang, C. Xu, C. Zhang, J. Xu, Z. Peng, and J. Pei. vchain+: Optimizing verifiable blockchain boolean range queries. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 1927–1940. IEEE, 2022.

[54] S. Wang, T. T. A. Dinh, Q. Lin, Z. Xie, M. Zhang, Q. Cai, G. Chen, B. C. Ooi, and P. Ruan. Forkbase: An efficient storage engine for blockchain and forkable applications. *Proceedings of the VLDB Endowment*, 11(10), 2018.

[55] L. Widick, I. Ranasinghe, R. Dantu, and S. Jonnada. Blockchain based authentication and authorization framework for remote collaboration systems. In *2019 IEEE 20th International Symposium on" A World of Wireless, Mobile and Multimedia Networks"(WoWMoM)*, pages 1–7. IEEE, 2019.

[56] C. Xu, C. Zhang, J. Xu, and J. Pei. Slimchain: scaling blockchain transactions through off-chain storage and parallel processing. *Proceedings of the VLDB Endowment*, 14(11):2314–2326, 2021.

[57] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis. Authenticated join processing in outsourced databases. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 5–18, 2009.

[58] A. A. Yavuz and P. Ning. Baf: An efficient publicly verifiable secure audit logging scheme for distributed systems. In *2009 Annual Computer Security Applications Conference*, pages 219–228. IEEE, 2009.

[59] N. Zahed Benisi, M. Aminian, and B. Javadi. Blockchain-based decentralized storage networks: A survey. *Journal of Network and Computer Applications*, 162:102656, 2020.

[60] S. Zawoad, A. K. Dutta, and R. Hasan. Seclaas: secure logging-as-a-service for cloud forensics. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 219–230, 2013.

[61] S. Zawoad, A. K. Dutta, and R. Hasan. Towards building forensics enabled cloud through secure logging-as-a-service. *IEEE Transactions on Dependable*

and Secure Computing, 13(2):148–162, 2015.

[62] C. Zhang, C. Xu, H. Wang, J. Xu, and B. Choi. Authenticated keyword search in scalable hybrid-storage blockchains. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 996–1007. IEEE, 2021.

[63] C. Zhang, C. Xu, J. Xu, Y. Tang, and B. Choi. Gemˆ 2-tree: A gas-efficient structure for authenticated range queries in blockchain. In *2019 IEEE 35th international conference on data engineering (ICDE)*, pages 842–853. IEEE, 2019.

[64] N. Zhang, J. Li, W. Lou, and Y. T. Hou. Privacyguard: Enforcing private data usage with blockchain and attested execution. In *Data privacy management, cryptocurrencies and blockchain technology*, pages 345–353. Springer, 2018.

[65] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vsql: Verifying arbitrary sql queries over dynamic outsourced databases. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 863–880. IEEE, 2017.

[66] Y. Zhang, J. Katz, and C. Papamanthou. Integridb: Verifiable sql for outsourced databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1480–1491, 2015.

[67] Z. Zheng, S. Xie, H.-N. Dai, W. Chen, X. Chen, J. Weng, and M. Imran. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems*, 105:475–491, 2020.

[68] Q. Zhou, H. Huang, Z. Zheng, and J. Bian. Solutions to scalability of blockchain: A survey. *Ieee Access*, 8:16440–16455, 2020.

[69] W. Zhou, Y. Cai, Y. Peng, S. Wang, K. Ma, and F. Li. Veridb: An sgx-based verifiable database. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2182–2194, 2021.

[70] K. R. Özyilmaz, M. Doğan, and A. Yurdakul. Idmob: Iot data marketplace on blockchain. In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 11–19, 2018.