TAP: Transparent and Privacy-Preserving Data Services

Daniel Reijsbergen and Aung Maw, *Singapore University of Technology and Design;* Zheng Yang, *Southwest University;* Tien Tuan Anh Dinh and Jianying Zhou, *Singapore University of Technology and Design*

https://www.usenix.org/conference/usenixsecurity23/presentation/reijsbergen

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

TAP: Transparent and Privacy-Preserving Data Services

Daniël Reijsbergen[†] Aung Maw[†] Zheng Yang[‡] Tien Tuan Anh Dinh[†] Jianying Zhou[†]

[†]*Singapore University of Technology and Design, Singapore, Singapore*

[‡]*Southwest University, Chongqing, China*

Abstract

Users today expect more security from services that handle their data. In addition to traditional data privacy and integrity requirements, they expect *transparency*, i.e., that the service's processing of the data is verifiable by users and trusted auditors. Our goal is to build a multi-user system that provides data privacy, integrity, and transparency for a large number of operations, while achieving practical performance.

To this end, we first identify the limitations of existing approaches that use *authenticated data structures*. We find that they fall into two categories: 1) those that hide each user's data from other users, but have a limited range of verifiable operations (e.g., CONIKS, Merkle^2, and Proofs of Liabilities), and 2) those that support a wide range of verifiable operations, but make all data publicly visible (e.g., IntegriDB and FalconDB). We then present TAP to address the above limitations. The key component of TAP is a novel tree data structure that supports efficient result verification, and relies on independent audits that use zero-knowledge range proofs to show that the tree is constructed correctly without revealing user data. TAP supports a broad range of verifiable operations, including quantiles and sample standard deviations. We conduct a comprehensive evaluation of TAP, and compare it against two state-of-the-art baselines, namely IntegriDB and Merkle^2, showing that the system is practical at scale.

*This research / project is supported by the National Research Foundation, Singapore, under its National Satellite of Excellence Programme "Design Science and Technology for Secure Critical Infrastructure" (Award Number: NSoE_DeST-SCI2019-0009), and by the National Research Foundation, Singapore, and Ministry of National Development, Singapore under its Cities of Tomorrow R&D Programme (CoT Award COT-V2-2020-1). Tien Tuan Anh Dinh is supported by the Singapore University of Technology and Design startup grant, SRG ISTD 2019 144. Zheng Yang is supported by the Natural Science Foundation of China (Grant No. 61872051). Jianying Zhou is supported by A*STAR under its RIE2020 Advanced Manufacturing and Engineering (AME) Industry Alignment Fund - Pre Positioning (IAF-PP) Award A19D6a0053. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, Ministry of National Development, Singapore, or A*STAR.

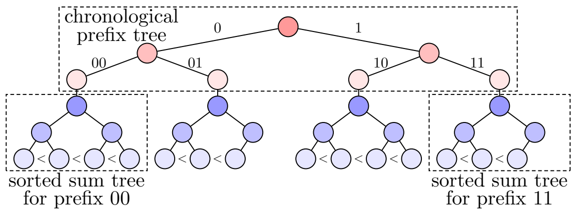

chronological prefix tree
0
1
00
01
10
11
sorted sum tree for prefix 00
sorted sum tree for prefix 11

Figure 1: TAP's authenticated data structure.

1 Introduction

Many of today's applications collect large amounts of user data and make decisions that have a direct impact on the user. One example is a utility company that collects power usage data from users and charges them different rates based on peak/off-peak periods per region. Another example is a road pricing system that determines real-time traffic conditions based on the number of cars on the road, and charges motorists appropriate congestion fees. The third example is an advertising service that monitors clicks and pays the publisher for displaying the advertisement based on the click-through rate. One desirable property of the applications above is *transparency* [12, 27], which allows users to verify that the computation done on their data has been executed correctly. This property is stronger than simply ensuring data *integrity*, as it protects users from malicious or compromised service providers who ignore data or tamper with the computation.

One approach toward transparent data services is for the provider to make the raw data available to users. For example, the provider can use a public bulletin board to store the data. Users can then perform computations directly on the data, but they have no data *privacy*. This is unacceptable in applications such as dynamic road pricing, as it would allow any user to track the movements of other users using the raw data. Alternatively, the service provider could store the data and return only query results. For example, an advertising service may have an API that returns the total number of clicks on an advertisement over a given period. However, this approach cannot guarantee transparency, which is troublesome when the provider has a financial incentive to falsify query results.

Our goal is to build a data service system with the following properties. First, it supports a wide range of queries that compute *aggregates* over the data of many independent users. Second, it protects data privacy from adversarial users. Third, it protects data integrity and transparency from an adversarial provider. Finally, it has a reasonable *performance* overhead, and scales well when the number of users increases.

One important primitive for realizing our goal is the *Authenticated Data Structure* (ADS) [33], which allows users to detect incorrect results returned by the provider. However, all ADS proposals in the literature fall into two broad categories, which both fall short in meeting our requirements. The first category consists of approaches that guarantee privacy, but support a limited set of queries. This category includes *transparency logs* such as certificate transparency [20] and its variants such as revocation transparency [21], extended certificate transparency [31], Trillian [16], CONIKS [25], and Merkle^2 [18]. Transparency logs store data in the form of key-value tuples and allow for public auditing. However, transparency logs only support data insertion, removal, and look-up operations, and thus fail to meet our first requirement. Also included in the first category are Proofs of Liabilities (PoLs) [13, 19], which allow users to prove that certain sums of user values are non-negative without revealing the underlying values. The second category consists of SQL-based authenticated databases such as IntegriDB [35] and FalconDB [28]. These databases allow users to verify the results of a wide range of SQL queries – i.e., sum, count, average, min, and max. However, they are either designed for a single user [35], thus failing our first requirement, or make all insert queries public [28], thus violating the second requirement. Other approaches exist that only support transparent range queries [12, 27, 29] and hence fail the first requirement.

In this work, we present a Transparent and Privacy-Preserving ADS named *TAP*, a data service system that meets our four design goals. It addresses the limitations of existing systems with a novel tree data structure, depicted in Figure 1, that combines the features of existing approaches that are best suited to our context. The data structure consists of a chronological prefix tree (as in Merkle^2), and each leaf in the prefix tree is the root of a Merkle sum tree (as in PoLs) that is sorted (as in IntegriDB). TAP adopts the same system model as CONIKS and Certificate Transparency, in which individual users *monitor* the inclusion of their data, and there exist some *auditors* that verify the relevant properties – e.g., ordered and append-only – of the data structure. The prefix trees enable efficient monitoring and auditing, similar to the data structures of CONIKS and Merkle^2. Meanwhile, the sorted sum trees enable fast verification for a wide range of operations, including sum, count, average, min, max, and sample standard deviation queries. TAP also supports the quantile query [22] that allows for the computation of fine-grained statistics, e.g., the median or the 5th percentile on sliding windows.

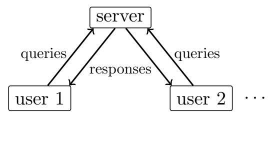
TAP is designed to store data from multiple users, thus meeting the first requirement. It protects data privacy – the second requirement – by storing cryptographic commitments instead of raw data, and by publishing zero-knowledge proofs. TAP's Merkle tree structure ensures data integrity and allows users to verify the correctness of a broad range of queries – the third requirement – by generating Merkle proofs and zero-knowledge range proofs for the commitments' underlying values. In our setting – i.e., a single data table in which aggregates are computed over sliding windows – TAP has better performance than previous systems, because it maintains one single tree, as opposed to the many trees in IntegriDB, and the tree is smaller than the tree in Merkle^2. The computation and bandwidth overheads of TAP are linear in the size of the sliding windows, but logarithmic in the size of the entire tree.

To reason about the security of TAP, we require the following: that each user adds one data entry per time slot, that the set of users (but not their data) at each time slot is known to a super-auditor (e.g., a regulator or watchdog), and that the fraction of adversarial users is bounded. We present a detailed analysis of the properties of TAP in this setting, and find an explicit tradeoff between transparency and privacy. We focus on guaranteeing perfect transparency at the cost of revealing query results, which cannot be trivially linked to user identities. In Appendix C, we discuss a method that guarantees (ε, δ)-differential privacy [14]. Our final contribution is a full, publicly accessible implementation of TAP, and we conduct a broad range of experiments to evaluate its performance. We compare TAP against two relevant baselines – Merkle^2 and IntegriDB. The results show that the system has reasonable overhead, and that it outperforms the baselines in many cases.

Contributions. We make the following contributions:

1. We present a survey of existing ADS approaches, and discuss their limitations in today's emerging applications.
2. We present TAP, a multi-user data service whose ADS combines elements from CONIKS, Merkle^2, IntegriDB, and PoLs to protect data privacy and integrity, while providing transparency to a wide range of operations.
3. We formally analyze TAP and prove that it only reveals the results of queries.
4. We present a full implementation of TAP and evaluate its performance. We compare it against two baselines, namely IntegriDB and Merkle^2, and show that TAP outperforms the baselines in many cases.

Outline. The remainder of this work is organized as follows. Section 2 describes the system model, use case examples, threat model, and our design goals. Section 3 discusses related systems built on top of ADSs. Section 4 presents TAP. Section 5 provides security and performance analysis of TAP, and discusses its current limitations. Section 6 contains the



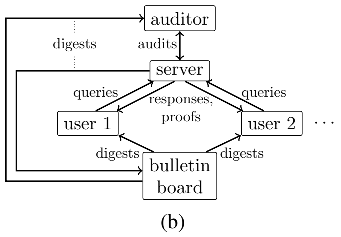


Figure 2: Left: system model with a trusted server. Right: TAP's system model with an untrusted server.

detailed performance evaluation and comparison against two state-of-the-art baselines. We discuss the practical aspects of TAP in Section 7, and Section 8 concludes the paper.

2 Model & Requirements

In this section, we first discuss the general system and data models. Next, we present three use cases for transparent data services and discuss how they fit into our models. Finally, we present the threat model and system requirements.

2.1 Model Entities

Our system model consists of the following types of entities:

Users. Users send data to the server and issue *queries* on the aggregate data through a client. Each user *monitors* the data structure by verifying that her data is properly stored by the server and verifies that query results are computed correctly. In practice, monitoring can be automated, e.g., the app on the user's mobile phone that shows bills or payments can verify the displayed values by querying the server's ADS.

Server. The server stores the data provided by the users in a database, and maintains an ADS on top of the data. It computes *responses* to user queries, and generates *proofs* for the responses using the ADS.

Auditors. Auditors validate the server's ADS. In particular, they verify that it is *append-only*, i.e., data is never modified or deleted, and that certain data has been *sorted* correctly. We will discuss these checks in more detail in later sections.

Bulletin board. The server periodically publishes the *digest* of its ADS to an immutable bulletin board, e.g., a public blockchain. Users and auditors download the latest digests during monitoring, auditing, and query verification. The only goal of the bulletin board is to prevent equivocation – i.e., the server presenting different versions of the ADS to different entities – and can therefore be replaced by a gossip protocol that would serve the same purpose.

Figure 2a displays a naïve design which assumes that the server is fully trusted, hence allowing an unscrupulous operator to return incorrect query results. Figure 2b displays our system model, in which the server is untrusted.

2.2 Data Model

We consider a simple relational data model. The schema consists of the following attributes.

Time. Time is modeled as a sequence of *epochs*, i.e., time slots such as hours or days, and is represented as an integer.

Value. This attribute contains the privacy-sensitive data. For simplicity, we assume that they are non-negative integers.

ID. This represents the user ID associated with the value.

Type. These are string attributes that capture metadata about users.

We denote the number of Type attributes by m, and the total number of attributes in our schema by $m' = m+3$. We assume that there is one data entry per ID per epoch. This assumption prevents a single adversarial user from arbitrarily skewing query results, as we will discuss in Section 5. However, a single entity may control multiple IDs, e.g., a single person owning multiple cars for congestion pricing.

2.3 Use Case Examples

In the following, we present three illustrative use cases for a transparent data service. In each case, the users receive *bills* or *rewards* that are dependent on the recorded data, including (potentially) the data of others. In all cases, the Time attribute is the smallest time interval in which the billing or reward rate remains the same. In the following, we discuss the key features of each use case, including the ID, Value and Type attributes, the typical scale of the system, and types of queries.

Smart Grids. Our first example is a smart grid that enables *peak/off-peak pricing*, i.e., customers are billed at a higher rate for power usage when system-level demand is high. In this setting, each *ID* corresponds to a smart meter's serial number, and the *Value* to the customer's power usage during the epoch. Possible *Type* attributes include the customer's geographic location, and the customer's type (industrial, commercial, or residential). The service is maintained by the power retailer. To illustrate the size of a typical smart grid, we consider SP Group, which is the largest power retailer in Singapore with 1.6 million customers in 2022 [6]. For peak/off-peak pricing, we record power usage once per hour. For the Type attributes, we consider the 28 postal code regions of Singapore, and the 3 customer types mentioned above.

In this setting, the main design goal for a transparent data service is to allow customers to verify their electricity bills. Each customer's bill for a given period T can be expressed as the sum, over each epoch t in T, of the electricity price in t multiplied by the customer's power usage in t. The first advantage of our data service is that it allows customers to verify their usage and bills. However, a major advantage of TAP is that it also enables more advanced pricing methods, such as making the price dependent on the *system-level* power usage [30]. The data service allows the users to verify claims about the system-level through sum queries. Finally, the data service allows for the computation of aggregate statistics, e.g., 1) the average and standard deviation of power consumption of all users within the same region, 2) the maximum and minimum usage in a region during a given period, and 3) the top 5% consumption across all residential users.

Congestion Pricing. Our second example is a system in which vehicles are charged when they cross designated road sections that are heavily congested during peak periods. To detect vehicles, *gantries* with cameras are placed alongside the designated road sections. Each *ID* corresponds to a *pair* of vehicle and gantry IDs, and the *Value* to the number of times the vehicle crossed the gantry during the epoch. Possible *Type* attributes include the gantry ID and the type of vehicle, e.g., car, truck, or motorcycle. As an illustrative example, we consider the Electronic Road Pricing (ERP) system in Singapore. As of 2022, the ERP system consists of 77 gantries of which around 20 are located at the entries and exits of the highly congested Central Business District (CBD) [2].

As in smart grids, the data service allows users to verify their bills even for advanced pricing schemes. For example, the price can be made dependent on the total number of registered vehicles that have crossed a gantry in an epoch, which would act as a proxy for the real congestion in the system. An even more advanced pricing scheme would make the price of entering the CBD dependent on the sum of vehicle entries into the CBD minus the sum of vehicle exits.

Digital Advertising. Our final example is a system that rewards websites who display digital advertisements [8]. In particular, a website owner receives a reward whenever an advertisement displayed on the website is clicked.[1] In this setting, *ID* corresponds to a pair of website and ad IDs, and the *Value* to the number of times the advertisement was clicked on via the website (i.e., the click-through rate). Possible Type attributes include the category and size of the website, or characteristics of the advertisement. The advertisement platform operates the server, while the website owners and advertisers monitor their data entries. The biggest online advertisement platform is Google Ads with millions of registered websites and advertisers. However, there are also digital ad platforms such as sixads [5] that serve around 100 000 websites.

The data service allows both the advertisers and the websites to monitor the click-through data provided by the advertising platform. This makes it easier to detect misbehavior, e.g., the platform underreporting (or overreporting) the number of clicks to the website owner (or advertiser) for profit. Finally, it enables more advanced reward schemes, e.g., in which the total reward paid by any single advertiser is limited.

2.4 Threat Model

We consider two types of threats. The first consists of *honest-but-curious* users who follow the protocol but try to learn the privacy-sensitive data of other users. The second is an *adversarial server* who tries to falsify query results by tampering with the data and/or query execution. This assumption captures unscrupulous server owners, insider threats, and external attacks via software vulnerabilities. We do not consider threats to privacy that stem from collusion between the server and users, because the server is always free to send raw data to adversarial users out-of-protocol. However, a limited set of adversarial users may collude with the server to falsify query results – this includes fake users created by the server. We discuss how users disseminate information about server misbehavior in Section 7.

We assume that all communication between the server and honest users is done via secure channels. The auditors are only trusted to validate the server's structural properties of the server's ADS, but not individual data entries. In fact, the trust assumption for auditors is the same as for users, i.e., any user with large computation and network resources can also act as an auditor. In practice, we also assume that one "super" auditor (e.g., a regulator) has the ability to verify user identities, to prevent service providers from creating an unlimited number of fake users. Finally, we assume that the public bulletin board – or, alternatively, the gossip protocol for users – is trusted and able to detect equivocation.

2.5 Requirements

Our goal is to design a system that meets the following requirements. Due to space constraints, we only present informal definitions of the security requirements, and leave the formal definitions to Section 5 and the appendix.

(R1) Rich operations for multiple users. The system supports a wide range of operations (or queries) on the aggregate data generated by multiple independent users.

(R2) Data privacy. A user can only learn a limited number of other users' values by performing queries.

(R3a) Data integrity. The server cannot change the data without being detected.

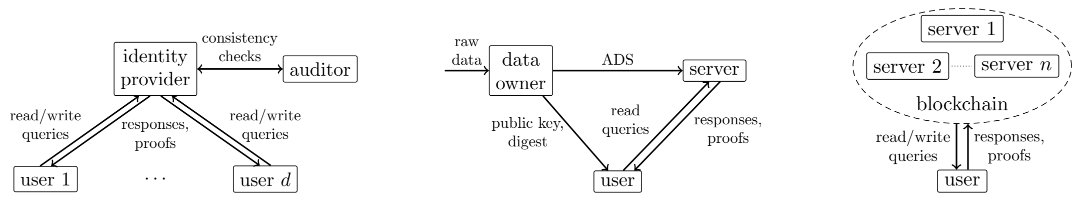
(R3b) Transparency. For each supported query, the server cannot convince the user to accept incorrect results computed from incorrect, incomplete, or artificial data.

(R4) Efficiency. The computation, storage, and network costs at the server and the user's client are small. Query overheads grow sublinearly with the number of users and epochs.

3 Existing Solutions

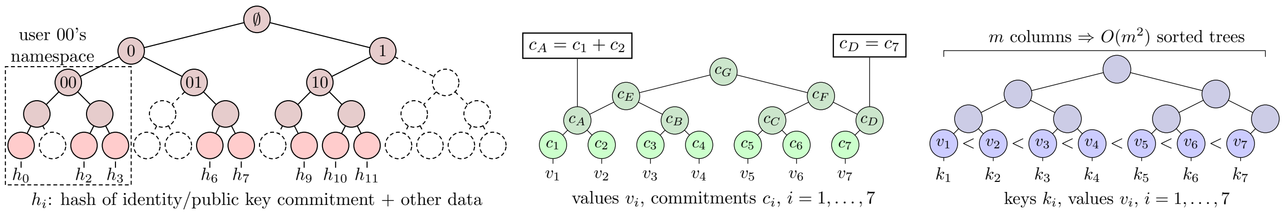
In this section, we discuss existing approaches that partially meet our requirements. We divide them into three categories: transparency logs, PoLs, and SQL-based authenticated databases. Visual representations of various system models, and several ADS designs whose features are incorporated in TAP, can be found in Figure 4. We conclude the section by discussing which requirements are met by these approaches.

[1]This is a simplified version of online advertising, as advertisers and publishers are often interested in more detail than just click-through rates.



(a) CONIKS (b) IntegriDB (c) FalconDB

Figure 3: System models of CONIKS, IntegriDB, and FalconDB.


user 00's namespace
$\emptyset$
0
1
00
01
10
h_0 h_2 h_3 h_6 h_7 h_9 h_{10} h_{11}
h_i: hash of identity/public key commitment + other data
$c_A = c_1 + c_2$
$c_D = c_7$
c_G c_E c_F c_A c_B c_C c_D
c_1 c_2 c_3 c_4 c_5 c_6 c_7
v_1 v_2 v_3 v_4 v_5 v_6 v_7
values v_i, commitments c_i, $i = 1, \dots, 7$
m columns $\Rightarrow O(m^2)$ sorted trees
$v_1 < v_2 < v_3 < v_4 < v_5 < v_6 < v_7$
k_1 k_2 k_3 k_4 k_5 k_6 k_7
keys k_i, values v_i, $i = 1, \dots, 7$

(a) CONIKS (b) Proofs-of-Liabilities (c) IntegriDB and FalconDB

Figure 4: ADS designs of CONIKS, Proofs-of-Liabilities, IntegriDB, and FalconDB.

3.1 Transparency Logs

Transparency logs are append-only data structures whose integrity is protected by a Merkle tree. They provide efficient cryptographic proofs that show, e.g., that an entry is included in the log, or that one log is a prefix of another log.

Certificate Transparency (CT). CT [20] addresses the problem of compromised certificate authorities by publishing the certificates on a transparency log. The system model of CT consists of some certificate authorities who issue certificates and insert them into the log, and users who search for specific certificates in the log. CT relies on a monitor to ensure the consistency of the log. The core data structure is a Merkle tree in which the certificates are hashed and stored at the leaves, and the server signs the root of the tree. CT has been extended to support efficient verification of certificate revocations [21, 31]. It has also been generalized into an abstraction called a *verifiable log* which is implemented as Google's Trillian [16].

CONIKS. CONIKS [25] extends CT to support transparent name-to-key bindings. It allows for efficient proofs of non-inclusion so that users can easily check for unauthorized name-to-key bindings in their namespaces. CONIKS' system model, depicted in Figure 3a for a system with d users, is similar to that of CT, but users are more active in monitoring their key bindings. CONIKS uses *prefix trees*, depicted in Figure 4a, for efficient non-inclusion proofs. It hides bindings by storing only their commitments at the leaves. It also hides the total number of users by adding dummy nodes.

Merkle^2. Merkle^2 [18] extends CONIKS through a data structure that enables efficient auditing. The data structure combines a *chronological Merkle tree* with prefix trees. The leaves of the chronological tree only extend to the right (append-only). Each internal node protects a set of leaves in the chronological tree, and stores the root of a prefix tree that has the same set of leaves.

3.2 Proofs of Liabilities (PoLs)

PoLs [13] are designed to prove solvency – i.e., assets being greater than liabilities – in a setting where users are fully anonymous and their individual assets and liabilities are privacy-sensitive. The main data structure in PoLs is a *Merkle sum tree*, as depicted in Figure 4b, that stores additively homomorphic *commitments* to the values of assets and liabilities in the leaves. Intermediate nodes store the sums of the commitments in their children. To show that the sum of assets and liabilities in the leaves is non-negative, PoLs use *zero-knowledge range proofs*. PoLs were generalized in [19] to use cases beyond proving solvency.

3.3 SQL-Based Authenticated Databases

IntegriDB. The system model of IntegriDB [35] is designed for outsourced databases. In particular, the user uploads data and metadata to an untrusted server, depicted in Figure 3b. The server executes user queries and generates proofs based on the metadata to show that the results are correct. IntegriDB supports data insertions, join queries on multiple tables, multidimensional range queries, and sum, count, average, min, and max queries. IntegriDB creates a sorted tree for each column pair, resulting in $\frac{1}{2}(m^2 - m)$ trees for a table with m columns, as depicted in Figure 4c. In each internal node of the tree, IntegriDB stores a polynomial over the values in the leaves of the internal node's subtree. The polynomials enable proofs that sum or range queries have been performed over the correct set of leaves. Meanwhile, the sorted nature of the trees allows users to verify min and max queries. Another SQL-based ADS, vSQL [34], supports generic SQL queries and has similar performance as IntegriDB.

FalconDB. FalconDB [28] combines IntegriDB with blockchains. In FalconDB's system model, depicted in Figure 3c, a smart contract maintains the ADS and ensures that it is globally consistent. Queries are performed directly by the servers, which run IntegriDB, without going through consensus (except for insertions and removals). Users verify the results by checking that the ADSs at the servers are the same as in the blockchain.

3.4 Limitations of Existing Solutions

Transparency logs meet our requirements of multiple user support, privacy, integrity, and efficiency. However, the range of supported operations, namely insertion, deletion, inclusion, and non-inclusion, is too limited to achieve R1. Similarly, PoLs do not achieve R1 as they only support sum queries, but not standard deviations, minima/maxima, or quantiles.

IntegriDB and Falcon achieve R3a, R3b, and R4, but cannot support R1 and R2 simultaneously. IntegriDB's system model assumes that a single user generates the ADS correctly before uploading it to the server. As such, IntegriDB cannot be easily extended to support multiple independent users. In particular, the server can maintain separate databases and ADSs, but users cannot verify operations on the aggregate data without building the data structures on the entire data by themselves. The users must therefore know each other's data, i.e., the system cannot simultaxneously achieve R1 and R2. FalconDB supports multiple users, but all data are stored on the blockchain, thus it does not meet R2.

4 TAP

In this section, we describe TAP, a transparent data service that overcomes the limitations discussed in Section 3.4. We start by describing the main building blocks, and then explain the core ADS and how it supports a rich set of operations. Finally, we discuss how audits are performed in TAP.

4.1 Preliminaries

We use several cryptographic primitives. We only define them briefly here due to space constraints, and refer readers to the literature for their formal and complete definitions.

A *hash function* H takes as input a value $x \in \{0,1\}^*$ and outputs a value in $\{0,1\}^{l_H}$, where l_H is the output length of the hash function. The function is *collision-resistant* if the probability of finding two different inputs that produce the same hash output is negligible.

A *commitment scheme* COM consists of two algorithms. COM.SETUP takes as input a security parameter 1^κ and outputs the commitment parameters P_c. Let V_c and R_c be the sets of all possible data and random values, respectively. COM.COMMIT takes as input the parameters P_c, a value $v \in V_c$, and a random value $r \in R_c$ (which we also call a *seed* to avoid confusion with data values), and outputs a commitment c'. COM is called *hiding* when c' reveals nothing about v, and *binding* when given a commitment of v and r, it is computationally infeasible to find another v' and r' that produce the same commitment. We use the shorthand notation $C(v,r) = \text{COM.COMMIT}(P_c, v, r)$, with P_c set during initialization. The scheme is *additively homomorphic* if for any $v_0, v_1 \in V_c$ and $r_0, r_1 \in R_c$, it holds that

$$C(v_0,r_0) + C(v_1,r_1) = C(v_0+v_1, r_0+r_1).$$

A *non-interactive zero-knowledge proof system* consists of three algorithms. NIZK.SETUP takes as input a security parameter 1^κ and outputs system parameters P_{zk}. NIZK.PROVE takes as input the system parameters P_{zk} and a statement-witness pair (s,w) and outputs a proof π. NIZK.VERIFY takes as input P_{zk}, a statement s, and a proof π, and outputs TRUE or FALSE. The proof system NIZK satisfies *zero-knowledge* if the generated proofs reveal nothing about the witnesses, and *simulation-extractability* if for any proof generated by the adversary, there exists an efficient algorithm to extract the corresponding witnesses with a trap door. In TAP, we use NIZK over the following relation for the range proofs:

$$R_{zk}(v_{\max}) = \{(c, v_{\max}), (v,r), | c = C(v,r) \wedge v \in [0, v_{\max})\}$$

We can prove statements of the form $v \in [a,b]$ by proving $v - b + K \in [0,K]$ and $v - a \in [0,K]$ for large K [11].

A *Merkle tree* is a binary tree in which each node i stores a hash value h_i. The hashes have the following structure. The leaves contain the hashes of the values stored in the tree. For the internal nodes it holds that $h_i = H(h_{\text{LEFT}(i)} | h_{\text{RIGHT}(i)})$, where $|$ represents concatenation, $\text{LEFT}(i)$ and $\text{RIGHT}(i)$ respectively return the left and right child of i (if there is no child, they return 0), and $h_0 = (0)^{l_H}$. Similarly, a Merkle *sum tree* contains in each leaf i a commitment $c_i = C(v,r)$ where v is the leaf's value and r its seed, and each internal node i stores $c_i = c_{\text{LEFT}(i)} + c_{\text{RIGHT}(i)}$. Inclusion of a leaf node i in a Merkle tree can be proven through a *co-path*, i.e., the sibling nodes on the path between i and the tree's root. The prover can use the co-path and the leaf to rebuild the hash (or commitment) in the root, and compare it to its known value.

A *prefix tree* is a binary tree in which each leaf corresponds to a key-value pair (χ, ϕ) where χ is a bit string of length l_P. Each internal node i stores the key χ_i of length l', such that $l' < l_P$, and its left and right child nodes have the key $\chi_i|0$ and $\chi_i|1$, respectively. A prefix tree is typically extremely sparse, and we only need to store the internal nodes that are on a direct path between a leaf and the root (which has key $\emptyset$). A prefix tree can be extended to a Merkle prefix tree by including a hash of the last prefix bit and value in each leaf, and the hash of the last bit and child hashes in internal nodes.

Time	ID	Type	Value
0	Alice	residential	11
0	Bob	residential	24
0	Carol	residential	13
1	Alice	residential	19
1	Bob	residential	26
1	Carol	residential	27
1	Dave	residential	26
1	Erin	industrial	36

Table 1: Illustrative data for Figure 5.

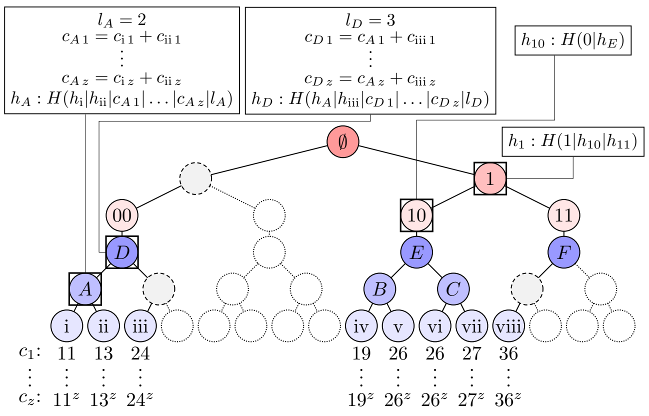
4.2 Data Structure

The ADS in TAP combines a single Merkle prefix tree (as in Figure 4a) with multiple sorted Merkle sum trees (which combine the key features of Figures 4b and 4c). The *key* of each leaf i in the prefix tree corresponds to a unique combination of values of the Time and Type attributes, whereas the *value* of i is the root hash of a sorted Merkle sum tree. This Merkle sum tree is constructed from the Value attributes of the data whose Time and Type attributes are equal to the prefix tree leaf's key. The tree is sorted by the values of its leaves in ascending order. Each leaf not only stores the raw value v, but also $v^2, v^3, \ldots, v^z$ for some system-wide integer z. These values enable the computation of advanced statistics (e.g., the standard deviation), while having no impact on the ordering because x^k is monotonically non-decreasing for $x, k \geq 0$.

The prefix tree is stored in memory, whereas the full table is stored in a SQL database. We do not keep the Merkle sum trees in memory except for their root hashes, which are stored in the prefix tree leaves, because the Merkle sum trees are easily constructed on-the-fly during queries.

The data structure in TAP is initialized as follows. For simplicity, we assume that the server takes as input a table consisting of initial data from multiple users. Given a table with Time, ID, Type, Value attributes, the server generates a random seed r_i for each row i and stores it as a new attribute **Seed**. The server then computes the sets of all *unique combinations* of values of the Time and Type attributes; we denote this set by $\mathcal{S}$. For each unique value tuple $s \in \mathcal{S}$, we determine the array $V_s = (v_{s\,i})_{i=1,\ldots,|V_s|}$ that contains the Value attributes of the rows whose Time and Type attributes match the elements of s. The server sorts V_s in ascending order, i.e., $v_{s\,i} \leq v_{s\,i+1}$ for all $i = 1, \ldots, |V_s| - 1$. Let u_i be the ID attribute of the row with corresponding Value v_i. For each $s \in \mathcal{S}$ such that $|V_s| > 0$, the server creates a Merkle sum tree where the ith leaf contains the following values: 1) $c_{i\,j} = C(v_i^j, r_i)$ for $j \leq z$, where $v_i \in V_s$ and r_i is the corresponding Seed attribute, 2) $l_i = 1$, and 3) $h_i = H(c_{i1}|\ldots|c_{iz}|H(u_i, s_0))$. For the empty node 0, we choose $c_0 = C(0,0)$, $l_0 = 0$, and $h_0 = H(0)$. Informally, $c_{i\,j}$ contains the commitment of the ith value (in ascending order) to the power j, l_i contains the *count*, i.e., the number of leaves in the subtree rooted at the node (which is always 1 for the leaf nodes), and h_i contains the leaf's hash.

Each internal node i contains the following values:



$l_A = 2$
$c_{A\,1} = c_{\mathrm{i}\,1} + c_{\mathrm{ii}\,1}$
$c_{A\,z} = c_{\mathrm{i}\,z} + c_{\mathrm{ii}\,z}$
$h_A : H(h_{\mathrm{i}}|h_{\mathrm{ii}}|c_{A\,1}|\ldots|c_{A\,z}|l_A)$
$l_D = 3$
$c_{D\,1} = c_{A\,1} + c_{\mathrm{iii}\,1}$
$c_{D\,z} = c_{A\,z} + c_{\mathrm{iii}\,z}$
$h_D : H(h_A|h_{\mathrm{iii}}|c_{D\,1}|\ldots|c_{D\,z}|l_D)$
$h_{10} : H(0|h_E)$
$h_1 : H(1|h_{10}|h_{11})$
$\emptyset$
00
1
10
11
A
B
C
D
E
F
i
ii
iii
iv
v
vi
vii
viii
c_1: 11 13 24 19 26 26 27 36
c_z: 11^z 13^z 24^z 19^z 26^z 26^z 27^z 36^z

Figure 5: TAP's ADS after inserting the 8 rows from Table 1.

1) $c_{i\,j} = c_{\mathrm{LEFT}(i)\,j} + c_{\mathrm{RIGHT}(i)\,j}$ 2) $l_i = l_{\mathrm{LEFT}(i)} + l_{\mathrm{RIGHT}(i)}$, and 3) $h_i = H(h_{\mathrm{LEFT}(i)}|h_{\mathrm{RIGHT}(i)}|c_{i\,1}|\ldots|c_{i\,z}|l_i)$. The hash in the root of the sum tree is stored as the value in the prefix tree leaf whose prefix is the bit concatenation of the Time and Type values. After initialization, the server sends the initial digest δ_0, which is the hash of the Merkle prefix tree root, to the bulletin board. For every value v_i that is inserted, the server sends the random value r_i in the corresponding commitment to the user, which the user can later use to verify the commitment.

For example, Figure 5 depicts the ADS after processing the rows from Table 1, if 'residential' and 'industrial' are mapped to 0 and 1, respectively. Each sum tree leaf node (denoted by i, ..., viii) contains the commitments $c_1 \ldots, c_z$ of a Value column entry, and h and $l = 1$ as discussed previously. The intermediate sum tree nodes $A, \ldots, F$ contain the values l, $c_1 \ldots, c_z$, and h computed from their child nodes. The prefix tree nodes $\emptyset, \ldots, 11$ contain hash values, and each prefix tree leaf corresponds to a unique combination of Time/Type values. In this case, $\mathcal{S} = \{(0,0), (0,1), (1,0), (1,1)\}$, but there are no rows for which Time and Type respectively equal 0 and 1 ('industrial') – as such, only three leaves (00, 10, and 11) are stored in the prefix tree.

4.3 Queries

Insert. As mentioned in Section 2.2, only one new entry per user ID can be inserted into the tree per epoch. To insert the entries of epoch $t > 0$, the server computes $\mathcal{S}_t$, the set of unique combinations of values of the Type attributes, right-concatenated with the epoch t. It then determines the value sets V_s for each $s \in \mathcal{S}_t$, and constructs the Merkle sum trees whose leaves are the sorted values in V_s. Next, it inserts for each s for which $|V_s| > 0$ a new key-value pair (χ_s, ϕ_s) into the existing prefix tree, where χ_s is the the bit string that represents s, and ϕ_s is the root hash of the corresponding Merkle sum tree. The server sends the digest δ_t, i.e., the root of the updated Merkle prefix tree, to the bulletin board. Finally, it sends to each user i the random value r_i. Since the leading bits in the prefixes correspond to the Time attribute, the prefix tree is sorted chronologically. As in Merkle^2, this allows auditors to efficiency verify that the tree is append-only.

Look-up. The user can search for the value v tied to her ID and seed for a given specific Time attribute t. The server executes the query on the SQL database, and if it finds a result then it generates a proof $\pi = (v, \pi_1, \pi_2)$ as follows. First, it computes a prefix χ using t and the user's Type attributes. Next, it constructs the Merkle inclusion proof π_1 for the leaf (χ, ϕ) in the prefix tree, where ϕ is the value for key χ. It then produces another inclusion proof π_2 for the value $C(v,r)$ in the Merkle sum tree whose root is ϕ. The user then verifies the proof by requesting the digest δ_t from the bulletin board, computing $c = C(v,r)$, and verifying that the inclusion proofs are correct with respect to c and the digest.

If no data entry is found, then the server generates the following non-existence proof: $\pi = (\pi', N^*, (h'_i)_{i \in N^*})$ First, it computes a prefix χ as above, and constructs the Merkle inclusion proof π' for the leaf (χ, ϕ) in the prefix tree, where ϕ again is the value in the leaf with prefix χ. Let N^* be the set of leaves in the Merkle sum tree whose root is ϕ. For each node $i \in N^*$, the server also computes $h'_i = H(u_i|t)$. The user's client verifies the proof by checking the prefix tree inclusion proof, and checking for all leaves $i \in N^*$ that $h_i = H(c_{i1}|\dots|c_{iz}|h'_i)$. Finally, it rebuilds the sum tree root from the leaves and checks that it matches ϕ.

Range cover. Range queries are a subroutine of aggregate queries. Since TAP does not reveal individual data entries to unauthorized users, the server returns the set of prefix nodes that cover all entries in the specified range. In particular, the query contains $S^* = (t^{\min}, s_1^{\min}, \dots, s_m^{\min}, t^{\max}, s_1^{\max}, \dots, s_m^{\max})$ where $t^{\min}$ corresponds to the smallest value of the Time attribute in the range, $t^{\max}$ to the largest value, and $s_1^{\min}, s_2^{\min}, \dots$ and $s_1^{\max}, s_2^{\max}, \dots$ correspond to the Type attributes. The server returns a set containing each prefix node i for which it holds that the subtree rooted at i contains the data entries whose Time and Type attributes overlap with S^*.

To produce a proof, the server calls the function ExtendRangeProof as described in Algorithm 1 in Appendix A with the root of the prefix tree, $\emptyset$, and S^* as input arguments. This function recursively calls itself on the node's children. Given a set N', the client requests the digest δ_t from the bulletin board and verifies the following properties: N' is properly formed (i.e., all hashes follow from the child hashes), the root node included in N' has a hash that equals δ_t, and these nodes completely cover S^*.

Sum/Count/Average/Standard Deviation. A user can query the sum and count of the values over a given range S^* defined as above. The server executes the query and generates the following proof:

$$\pi = (N', L', r^*, v_1^*, \dots, v_z^*, (h'_i, c_{i1}, \dots, c_{iz}, l_i)_{i \in L'}).$$

In particular, it first computes the range cover proof N' by calling ExtendRangeProof from Algorithm 1 on the prefix

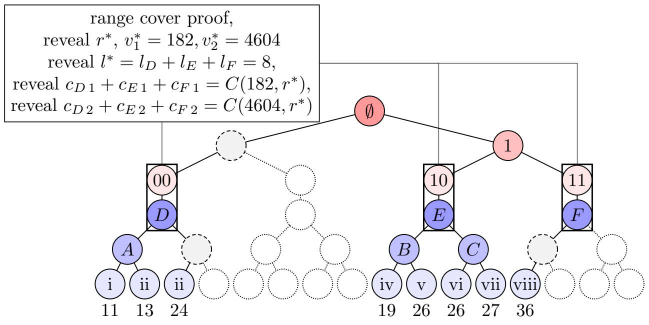

range cover proof,
reveal r^*, $v_1^* = 182, v_2^* = 4604$
reveal $l^* = l_D + l_E + l_F = 8$,
reveal $c_{D\,1} + c_{E\,1} + c_{F\,1} = C(182, r^*)$,
reveal $c_{D\,2} + c_{E\,2} + c_{F\,2} = C(4604, r^*)$
$\emptyset$
1
00
10
11
D
E
F
A
B
C
i
ii
ii
iv
v
vi
vii
viii
11
13
24
19
26
26
27
36

Figure 6: Example of a sum query in the ADS of Figure 5.

tree root. It then computes the sums $v_1^*, \dots, v_z^*$ and the total seed r^* of the covered data entries. Let L' be the set of leaves of the prefix tree in N'. The server then determines for each $i \in L'$ the child hash $h'_i = h_{\text{LEFT}(i)}|h_{\text{RIGHT}(i)}$, the commitments $c_{i1}, \dots, c_{iz}$, and the leaf count l_i.

Given π, the user's client first initializes $c_j^* = C(0,0)$ for $j = 1, \dots, z$, and $l^* = 0$. Next, it verifies the range cover proof N', then checks for every node $i \in L'$ whether it holds that

$$\phi(i) = H(h'_i|c_{i1}|\dots|c_{iz}|l_i),$$

i.e., whether $\phi(i)$, the value stored in prefix tree leaf i, is constructed as expected from the child hash, commitment, and leaf counts. If so, it updates c_j^* to $c_j^* + c_{nj}$ for all $j = 1, \dots, z$ and l^* to $l^* + l_i$. Finally, it checks that $c_j^* = C(v_j^*, r^*)$ for all $j = 1, \dots, z$. If so, the user can compute statistics such as the sum v_1^*, count l^*, and average v_1^*/l^*. It also enables the computation of more complex query results such as the sample standard deviation, i.e., $\sqrt{(v_2^* - (v_1^*)^2/l^*)/(l^* - 1)}$.

Figure 6 visualizes the response to a sum query over all values in Figure 5. The server first reveals the range proof, which is the entire prefix tree as the sum query covers the entire dataset. Next, the server reveals $l^* = 8$, $v_1^* = 182$, and $v_2^* = 4604$, which are verified by the client. This allows the client to compute, e.g., the average (22.75) and the sample standard deviation ($\approx$8.137).

Min/Max. A user can request the min value over a range S^* at epoch t, after which the server returns a proof

$$\pi = (N', L', v^*, i^*, (c_{i1}, \dots, c_{iz}, h'_i, \pi'_i, \pi_i^*)_{i \in L'}).$$

Here, N' is the proof of the range query, L' the set of prefix leaf nodes in N', v^* the minimum value in the range, and i^* the index of a prefix tree leaf whose sum tree contains v^*. For each node $i \in L'$, $c_{i1}, \dots, c_{iz}$ are the commitments of the value in the *leftmost* leaf in the sum tree of prefix tree leaf i – since the nodes are sorted, the leftmost leaf *must* have the smallest value in the sum tree. The hash h'_i equals the hash $H(u|t)$, such that u and t are the user ID and time in the leftmost leaf. Each inclusion proof π'_i, $i \in L'$, asserts that the sum tree leaf with hash $H(c_{i1}|\dots|c_{iz}|h'_i)$ is included in the sum tree whose root is stored in prefix tree leaf i. The zero-knowledge range proofs $(\pi_i^*)_{i \in L'}$ are as follows: if $i = i^*$, then the underlying value in the leftmost leaf must equal v^*, so π_i^* asserts that c_{i1}'s underlying value is in the range $[v^*, v^*+1)$.[2] Otherwise, π_i^* asserts that this value is in the range $[v^*, \infty)$.

Given π, the user's client first checks whether the range cover proof N' is correct. It then verifies for all $i \in L'$ that π_i' is a valid inclusion proof for the leaf with hash $H(c_{i1}|\dots|c_{iz}|h_i')$, and that this leaf is indeed the leftmost leaf in the path. Next, it verifies the range proofs. By treating i^* as a separate case, we guarantee that at least one sum tree contains the value v^* in the leftmost leaf. If all checks succeed, the user accepts v^* as the min value. The above is straightforwardly generalized to max queries by proving that the value in the *rightmost* sum tree leaves is at most equal to the maximum value v^*.

Quantile. A user can request a quantile by specifying a range S^* and a parameter $q \in [0,1]$ that indicates which quantile to compute. A value x is a q-quantile of a set V with n entries if there are at least nq entries in V whose value is at most x and at least $n(1-q)$ entries whose value is at least x. The server executes the query and returns the proof

$$\pi = (N', L', v^*, (\pi'_{\text{LEFT}\,i}, \pi^*_{\text{LEFT}\,i}, \pi'_{\text{RIGHT}\,i}, \pi^*_{\text{RIGHT}\,i})_{i \in L'}).$$

Here, N' is the proof of the range query, L' the set of leaf nodes in N', and v^* the result of the query. For each $i \in L'$, the server determines $\text{LEFTMOST}(i, v^*)$, which is the leftmost leaf in the sum tree corresponding to i whose value is at least v^*, if such a leaf exists. It computes the inclusion proof $\pi'_{\text{LEFT}\,i}$ for this leaf, and a zero-knowledge proof $\pi^*_{\text{LEFT}\,i}$ asserting that its underlying value is at least v^*. Next, it determines $\text{RIGHTMOST}(i, v^*)$, which is the rightmost leaf in i's sum tree whose value is at most v^*, if such a leaf exists. It then computes the inclusion proof $\pi'_{\text{LEFT}\,i}$ for this leaf, and a zero-knowledge proof $\pi^*_{\text{LEFT}\,i}$ asserting that its underlying value is at most v^*.

Given π, the user first checks that the range cover proof N' is correct. It then iterates over the nodes $i \in L'$, where L' is the set of leaf nodes in N', and verifies their inclusion and range proofs. The co-path of the Merkle tree inclusion proof includes the leaf count $l_{i'}$ for each node i' on the co-path; therefore, the user knows for each node on the co-path whether its leaves are to the right of $\text{LEFTMOST}(i, v^*)$. Let $L^{\text{LEFT}}(i)$ be the number of leaves to the right of node $\text{LEFTMOST}(i, v^*)$ in the sum tree. Let $L^{\text{RIGHT}}(i)$ be the number of leaves to the left of $\text{RIGHTMOST}(i, v^*)$ in the sum tree. The user verifies that $\sum_{i \in N'} L^{\text{LEFT}}(i) \geq nq$ and $\sum_{i \in N'} L^{\text{RIGHT}}(i) \geq n(1-q)$, and accepts the result v^* if the verification is successful.

Figure 7 visualizes the response to a query for the median (i.e., the $\frac{1}{2}$-quantile) over all values in Figure 5. In this case, the server can choose any value in the interval $[24, 26]$ as a valid median, e.g., $v^* = 26$. The server first shows that the rightmost leaf in the subtree for prefix '00', and the third leaf for prefix '10', have $v \leq 26$. The client uses the l-values in the inclusion proofs to determine that there are at least 4 nodes to

[2] Alternatively, the server can reveal the random value associated with v^*.

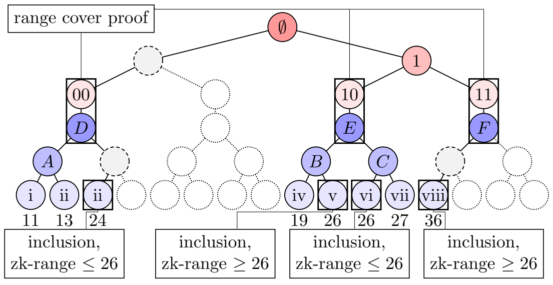


Figure 7: Example of a quantile query in the ADS of Figure 5.

the left of these nodes, which means that at least 6 leaves have $v \leq 26$. Finally, the server shows that the second leaf for prefix '10' and the leaf for prefix '11' have $v \geq 26$. As there are 2 leaves to the right of the former leaf, there are at least 4 leaves with $v \geq 26$, which proves that $v^* = 26$ is a valid median.

4.4 Auditing

The auditor requests the server to generate proofs that show that the tree is append-only and that the sum trees are sorted by the leaf values. For the proof that the tree at epoch t is constructed from the tree at epoch $t' < t$ in an append-only manner, the server includes each prefix tree leaf i whose Time attribute s_i' has $t' < s_i' \leq t$. Furthermore, it sends a set of internal nodes from the tree at t' such that the root of the tree at t can be rebuilt from the hashes in these nodes. (This is analogous to the procedure in Section V.A of [18].) The auditor then requests the digests $\delta_{t'}$ and δ_t from the bulletin board and checks that it can rebuild the two trees using the new leaves and the old internal nodes. To prove that the leaves in a sum tree with values $v_1, \dots, v_{n^*}$ are sorted, the server does the following. For each $i = 1, \dots, n^* - 1$, it generates a zero-knowledge range proof that $v_{i+1} - v_i \geq 0$ (this is possible because NIZK's underlying commitment scheme is additively homomorphic), and sends the proofs to the auditor.

5 Analysis

In this section, we discuss how TAP meets the requirements presented in Section 2.5. The server handles data from multiple users and supports a broad range of queries – i.e., all single-table queries supported by the baseline approaches, and additionally quantile queries. In order words, TAP satisfies requirement R1. In the following, we discuss TAP against requirements R2, R3a, R3b, and R4 in more detail. Due to space limitations, the formal security definitions can be found in the appendix, which we mention in the text when needed.

5.1 Security (R2, R3a, and R3b)

Privacy. We discuss the impact on privacy of each of the supported operations separately.

Look-up: an inclusion proof reveals a single value in a sum tree, but only if the user already knew that this value was included. A non-inclusion proof reveals a set of commitments.

Audit: reveals the prefix tree and the commitments in the sum trees, neither of which are privacy-sensitive.

Sum/count/average: reveals the sums over the values in the sum trees that correspond to the prefix leaf nodes in the requested range, but no individual values.

Min/max: reveals one unique value, namely the minimum/maximum value across the requested range, and the sum tree that contains this value.

Quantiles: reveals a value that may correspond to a single unique value in a tree, depending on the values in the requested range and the choice of the server (e.g., if the requested range contains the values $(1,2,3,4)$, then any value in the interval $[2,3]$ is a median, but if it contains $(1,2,2,3)$ then 2 is the unique median which is therefore revealed). However, it does not reveal which tree(s) (if any) contain this value.

TAP queries can leak other values in two cases: the sum if the number of values in a sum tree is very low, and the quantile query if a user is allowed to perform an unlimited number of queries. This follows from the following two theorems, which are proved in Appendix B.

Theorem 1. *If a sum tree has n leaves and f values are known by a coalition of users, then the remaining $n-f$ values can be decrypted if and only if $n-f<2$, regardless of many queries are executed.*

Theorem 2. *If a sum tree has n leaves and at most f arbitrary quantile queries can be performed, then at most f values can be decrypted.*

The privacy properties are proven in Appendix B through a formally defined *game*, $\mathsf{Game}^{\mathsf{Priv}}_{\mathsf{TAP},\mathcal{A}}$, in which the adversary $\mathcal{A}$ wins if she is able to distinguish the user behind the revealed values. Given Theorem 1 and 2, privacy can be achieved in practice if the server refuses to respond to sum queries over trees that have a limited number of leaves, and by restricting the range of values q over which q-quantiles can be performed. Since the server can efficiently prove claims about the number of leaves in the sum trees (which is the same as proving the result of a count query), the server cannot refuse to respond to valid queries without being detected. In this case, the sum and quantile queries reveal at best a limited number of values. The other query types either reveal one (min/max) or zero (count) values per query, so in this case a limited number of values are returned by default. Note that the querying language in TAP does not allow users to query sums of subsets within a single sum tree, so a sum query reveals nothing about a sum tree except the sum over *all* of its values. Finally, a prohibitively large number of look-up queries would be necessary to reveal a value that is unknown to the user, as the user would have to guess both the value and the random seed correctly.

In Appendix C, we discuss strategies to achieve stronger privacy at the expense of transparency by adding noise to

noise type	transparency	privacy
no noise / return true result	query result is always provably accurate	limited number of values are revealed
add bounded random noise	query result is accurate within some margin	(ε,δ)-differential privacy, $\delta>0$
add unbounded random noise	results can be arbitrarily distorted	provable ε-differential privacy

Table 2: Overview of the impact of adding noise to measurements on integrity and privacy.

query results. If this noise has a protocol-wide bound b, then the server can efficiently prove that the claimed result is no further than b away from the true value using zero-knowledge range proofs for all types of queries. If the noise is unbounded, then we can prove ε-differential privacy [15, 32]. However, unbounded noise allows a malicious server to arbitrarily distort results, which violates transparency (the threat model for differential privacy assumes that the server is honest). On the contrary, if the noise is bounded then the mechanism may satisfy the notion of (ε,δ)-differential privacy. A summary can be found in Table 2.

Integrity. Honest users monitor the inclusion of their data in each epoch by issuing look-up queries for their own entries and verifying the inclusion proofs. Therefore, if the server tries to add incorrect data for an honest user in an epoch, then it will be detected. Furthermore, the server cannot modify data from previous epochs, as this will cause the extension proof to fail during an audit. In Appendix B, this is formalized using the game $\mathsf{Game}^{\mathsf{Trans}}_{\mathsf{TAP},\mathcal{A}}$, in which the adversary wins if she able to convince the user to accept a modified value, and which is used to prove both integrity and transparency.

Transparency. For each type of query supported by TAP, the user can verify that the result is correct. For look-up queries, the user is provided with inclusion and non-inclusion proofs. For sum queries, the user is given inclusion and completeness proofs of the relevant nodes in the prefix tree, with which it can verify the sum by exploiting the additively homomorphic property of commitments in TAP. For min, max, and quantile proofs, the user is given zero-knowledge range proofs that assert that the committed value is greater than all, none, or a specified number of the total nodes in the specified range. As the cryptographic primitives are secure, the server cannot convince users to accept incorrect results.

For adversarial or fake users who collude with the server, the server is free to add arbitrary data. As such, sum, average, min, and max queries can be arbitrarily distorted: for example, if the true sum is v and the server wants to increase this to $v'>v$, then the server can select a single adversarial user i whose true value is v_i and increase it to $v'-v+v_i$. However, to distort a q-quantile query over a range with n values, a server would require collusion with at least $\max(q,1-q)n$ users. As such, quantile queries are inherently more robust than sum, min, and max queries. To mitigate the impact of adversarial users on sum queries, the server can make the decision to bound all individual values by an individual bound $\gamma > 0$. This may be of interest in use cases where extremely high values are unlikely – e.g., the smart grid or congestion pricing use cases of Section 2.3. In such, cases, the degree to which sum query results can be distorted is also limited: if f out of n users are adversarial, then the maximum possible distortion is $f\gamma$. Auditors can check that all individual measurements are below γ through zero-knowledge range proofs. A similar approach was recently proposed in [30].

5.2 Performance (R4)

Table 3 compares the asymptotic costs of queries in TAP to related systems. Here, n denotes the number of data rows, m the number of Type attributes (i.e., columns), d the number of users, and t the number of epochs (i.e., $n = O(td)$). For queries that compute an aggregate over a range, w denotes the number of epochs in the range.

Storage. For storage costs, we consider both the size of the ADS and the underlying dataset, which has size $O(mn)$. TAP stores only the prefix tree, which has $O(n)$ nodes, in memory and generates sum trees on-the-fly. The storage costs for TAP are smaller than those in the other systems except for CT. CONIKS regenerates the entire tree after each epoch, so the storage cost of the ADS is $O(tn)$ [18]. Merkle^2 uses a bigger tree than TAP because the internal nodes of Merkle^2's chronological tree store prefix trees. IntegriDB and FalconDB create a sorted tree for every combination of columns, leading to a total storage cost of $O(m^2n)$.

Insertion. Insertion is performed once per epoch with up to d entries. The server needs to insert $O(d)$ new leaves into the prefix tree and a commitment tree has to be built for each prefix tree leaf. Since the cost of inserting a new entry into the prefix tree is $O(\log n)$, the total cost is $O(d \log n)$. TAP has a higher insert cost than CT, CONIKS, and for reasonable values of n also than Merkle^2. However, TAP performs better than IntegriDB and FalconDB because they need to perform one insert for each of the m^2 trees in their ADS.

Inclusion proof. To prove that a data entry exists in the tree, the server has to generate an inclusion proof for a prefix tree leaf, generate the corresponding sum tree, and generate an inclusion proof of the data value in the sum tree. The cost of the first and third steps is $\log n$, and the worst-case cost of the second step is $O(d)$. Thus, the total cost is $O(d + \log n)$. CT, CONIKS, Merkle^2 are faster at generating the proof because they do not have to rebuild subtrees. In other words, TAP can improve at this cost by keeping all the sum trees in memory, at the cost of storage. However, we prioritize storage reduction by generating the sum trees on-the-fly, as look-ups are normally only performed once per user per epoch.

Non-inclusion proof. Non-inclusion proofs are similar to inclusion proofs, except that all leaves in the sum tree are included, of which there are d in the worst case. The asymptotic cost is therefore $O(d + \log n)$. TAP performs better than CT, IntegriDB, and FalconDB for this proof. However, it performs worse than CONIKS and Merkle^2. This is to be expected as non-inclusion proofs are heavily optimized in these two systems because it is one of their main use cases.

Auditing. To audit a single epoch, the auditor has to verify the append-only proof and range proofs for the sum trees. The former costs $O(d \log(n))$, and the latter costs $O(d)$. Therefore, the total cost is $O(d \log(n))$. Only CT and Merkle^2 have built-in support for audits. In CONIKS, the tree is rebuilt in each epoch, which makes auditing difficult. IntegriDB and FalconDB also do not support audits beyond checking all entries in the m^2 trees. Asymptotically, TAP is as efficient as CT and Merkle^2 – however, it requires verifying $O(d)$ zero-knowledge range proofs, which are considerably more expensive to verify than Merkle tree inclusion proofs.

Sum. To generate the proof for the sum query, the server first computes the range cover proof N'. The worst-case number of prefix tree leaves in N' is wd, as there are at most d leaves per epoch, and there are w epochs. For each leaf node in the range, its $\log(n)$ parents are included in N', Therefore, the cost of generating the proof is $O(wd \log n)$.

In IntegriDB and FalconDB, operations are performed on $m+1$ trees, one for each of the Type attributes and one for the Time attribute. In each tree, the sum is calculated from the polynomials stored in the internal nodes that form the minimal covering set of the leaves in the range. There are $O(\log(wd))$ of such internal nodes in total. The total cost is therefore at least $O(m \log(wd))$, although the results in Section 6 suggest that the processing costs at the server also depend on n, which is why the entries for the sum in IntegriDB and FalconDB are marked with an asterisk in Table 3.

Min/Max. The cost of min or max queries is similar to that of sum queries since they are based on the same range cover proof, except that for each sum tree the server also needs to generate a range proof and an inclusion proof. The asymptotic cost is still $O(wd \log n)$. For IntegriDB and FalconDB, the cost is the same as for the sum, i.e., $O(m \log(wd))$.

Quantile. This query requires at most two range and inclusion proofs per sum tree and a range cover proof – its asymptotic cost is therefore also $O(wd \log n)$, and no other baselines support this type of query.

6 Evaluation

In this section, we evaluate the practical performance of TAP. We first describe our implementation of TAP, then discuss our experimental setup, and finally present the empirical results. We conduct four types of experiments: microbenchmarks on a single machine, end-to-end experiments with the user and server on different machines, a performance comparison against two related baselines, and a scalability experiment that explores the limits of TAP. The final three sets of experiments were run on Amazon Web Services (AWS) EC2.

		asymptotic processing costs						
	storage	insert d rows in epoch	inclusion 1 row	non-inclusion 1 row	auditing 1 epoch	SUM	MIN/MAX	quantiles
CT	$O(mn)$	$O(d \log n)$	$O(\log n)$	$O(n)$	$O(d \log n)$	—	—	—
CONIKS	$O(mn+tn)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	—	—	—	—
Merkle^2	$O(nm+n\log n)$	$O(d \log^2 n)$	$O(\log^2 n)$	$O(\log^2 n)$	$O(\log n)$	—	—	—
IntegriDB	$O(m^2 n)$	$O(dm^2 \log n)$	$O(m^2 \log n)$	$O(m^2 n)$	—	$O(m \log(wd))^*$	$O(m \log(wd))^*$	—
FalconDB	$O(m^2 n)$	$O(dm^2 \log n)$	$O(m^2 \log n)$	$O(m^2 n)$	—	$O(m \log(wd))^*$	$O(m \log(wd))^*$	—
TAP	$O(mn)$	$O(d \log n)$	$O(d+\log n)$	$O(d+\log n)$	$O(d+\log n)$	$O(wd \log n)$	$O(wd \log n)$	$O(wd \log n)$

Table 3: Asymptotic costs of TAP versus other systems – here, n is the number of data entries, m the number of columns, d the number of users, t the number of epochs, and w the number of epochs in the queried range. The asterisks (*) indicate that although the number of tree nodes returned for the proofs of sum, min, and max queries in IntegriDB is sub-linear in d, this is not necessarily the case for the size of the data stored in those nodes.

operation	time cost (ms)	storage cost (B)
NIZK.Setup	11.450	22190
NIZK.Prove	134.780	74413
NIZK.Verify	70.040	—
generate commitment	0.144	162
sum two commitments	0.010	165

Table 4: Time and storage cost of cryptographic operations.

6.1 Implementation & Set-Up

We have fully implemented TAP in Go, and made the source code available at https://github.com/tap-group/transparent-data-service. We base our prefix tree implementation on Merkle^2. For commitments and zero-knowledge range proofs, we use the zkrp library from Morais et al. [1, 26]. This library uses Bulletproofs [10] for zero-knowledge range proofs and Pedersen commitment with the secp256k1 elliptic curve [9] for additively homomorphic commitments. We use Go's MySQL module for the database backend. We use the latest versions of the reference implementations of IntegriDB and Merkle^2 as of June 2022 [3, 4]. We use Merkle^2 as a baseline for look-up queries and audits, and IntegriDB – despite having a different system model and support for a broader range of SQL queries – for aggregate queries, as it is the most efficient approach with publicly available code that we are aware of. Although vSQL [34] reported comparable performance on a more general class of SQL queries than IntegriDB, we have not included vSQL as a baseline because its implementation is not publicly available.

The microbenchmarks were run on a MacBook Pro laptop with a 2.4 GHz Quad-Core Intel Core i5 processor and 16 GB of RAM, with iOS 11.6. For the end-to-end experiments, we ran the server on a *t2.xlarge* instance and the user on a *t2.micro* instance. For the comparison against IntegriDB and Merkle^2, we run all three systems on *t2.xlarge* instances. The scalability experiments were run on a 16-core *m5.4xlarge* instance. For some scalability experiments, we took the average over multiple runs to reduce the impact of random noise. To aid reproducibility, we have made an AWS virtual machine image that is set up for the scalability experiments publicly available with identifier ami-0935fdbddc542254e. The costs of various cryptographic operations in our system, when executed on the laptop, are shown in Table 4.

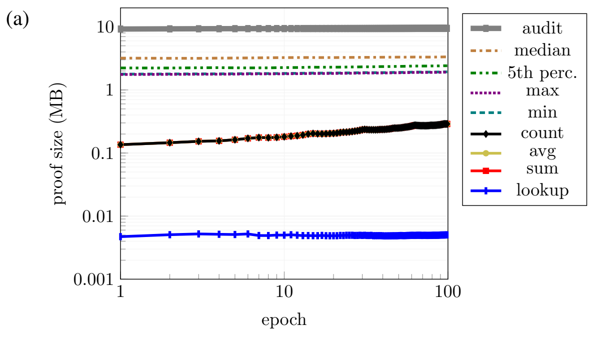
6.2 Microbenchmarks

We first evaluate the bandwidth costs of the different queries on a single machine. For this experiment, we consider $d = 100$ users who each insert a new value per epoch. We have two Type columns: *region* and *is_industrial*. Each user is randomly assigned to one of 10 regions, and 20% of the users have *is_industrial* set to 1 and the others have it set to 0.

Figure 8a compares the proof sizes of different queries, including audits. We observe several groups. The look-ups have the smallest proofs, as they only consist of two Merkle tree co-paths. They are followed by the sum, count, and average, whose proofs include all of the prefix tree leaves in the range query. Next are the min and max, which add range proofs for each sum tree. For quantile queries, we observe some difference between the median and the 5th percentile – the former has a larger proof size. The reason is that for each subtree, the query requires only a single inclusion and range proof if all of the sum tree's values are either greater or smaller than the quantile, and two inclusion and range proofs otherwise. The former scenario is more likely for the 5th percentile. The proof for audits is the largest, as it contains $O(d)$ range proofs. Even in this case, the size remains modest at 10MB per epoch.

6.3 End-to-End Performance

We measure end-to-end latencies of different user queries in a setting with $n = 1000$ and $m = 5$. Figure 8b shows the detailed breakdown. In particular, we divide the total time into five components: 1) Prefix tree proof generation: the time for generating the prefix tree inclusion proof for look-ups, and the range cover proof for the other queries. 2) Prefix tree



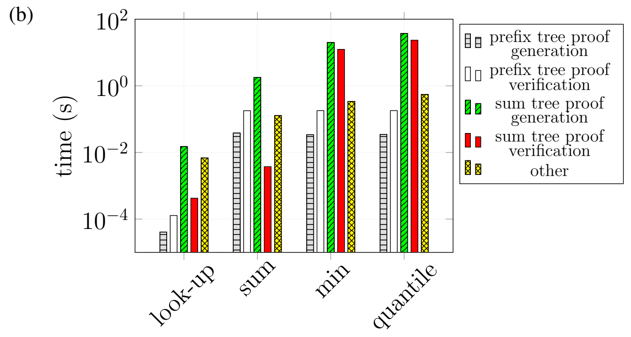


Figure 8: (a) Proof sizes of different query types, (b) Time breakdown for different query types.

proof verification: the time for verifying the prefix tree proofs. 3) Sum tree proof generation: the time for generating the sum tree inclusion proof for look-ups, the hashes needed to verify the commitments for sum queries, and the sum tree inclusion proof and range proofs for other queries. 4) Sum tree proof verification: the time for verifying the sum tree proofs. 5) Other: network delay and any other costs.

We observe that network delay is a significant factor in the end-to-end latency. As expected, this cost is proportional to the proof sizes. The remaining cost is dominated by the cost of rebuilding the sum trees. For sum queries, the range cover proofs have a large impact, but their costs are still an order of magnitude smaller than the sum tree proofs. The costs related to the sum tree proofs are especially large for min/max and quantile queries because they are dominated by the cost of generating and verifying zero-knowledge proofs. The end-to-end latencies for sum, min and quantile queries are $1s$, $23s$ and $60s$ respectively, which we believe to be reasonable for low-end virtual machines (*t2.xlarge* and *t2.micro*).

6.4 Comparison Against Baselines

To provide empirical evidence for the asymptotic cost differences shown in Table 3, we compare TAP against IntegriDB and Merkle^2 on different queries and with different data sizes. Figure 9a shows the storage costs for the three systems. Since TAP does not store the Merkle commitment trees but generates them on-the-fly during queries, it has the lowest storage cost. IntegriDB has the worst cost, as it needs to store at least 25 copies of a tree with the same number of leaves as Merkle^2.

In Figure 9b, we have displayed the audit costs of TAP versus Merkle^2. We omit IntegriDB from this graph as it does not support audits by default. It can be seen that the audits are orders of magnitude more expensive in TAP as in Merkle^2, which is due to the fact that an auditor in TAP has to evaluate dozens of zero-knowledge range proofs per epoch. However, we note that it takes only $60s$ to audit an epoch with 100 new entries on a low-end machine (*t2.xlarge*).

Figure 9c displays the costs of inserting 100 data entries in an epoch. Merkle^2 is faster than TAP, as the former does not need to generate sum trees on-the-fly. However, IntegriDB has the worst performance, because the new data needs to be inserted into at least 25 trees. The results for look-up queries, shown in Figure 9d, are similar to insertion queries, although the look-up costs for IntegriDB increase faster than the others. The IntegriDB reference implementation crashed when we performed queries on tables with more than 2000 rows.

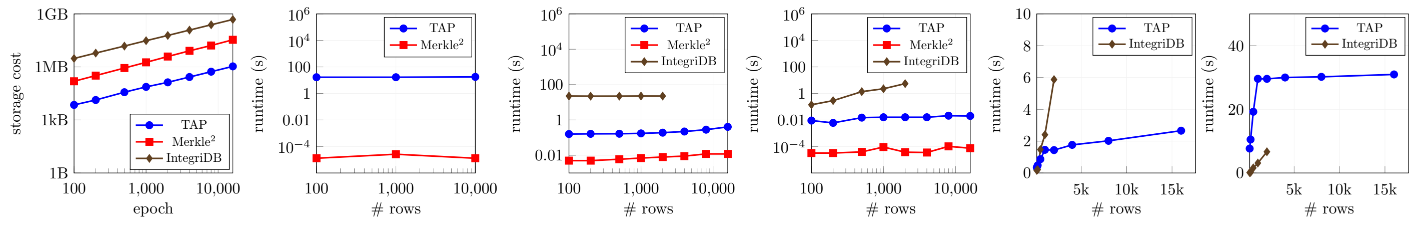
Figure 9e compares the cost of a sum query on the first 10 epochs for both TAP and IntegriDB. Although the proof size in IntegriDB depends only on the minimal covering sets of the values that contribute to the sum, we observe that its overall cost appears linear in the number of rows. We observe the same for min queries, as shown in Figure 9f. By extrapolating the line in Figure 9f beyond the point where the IntegriDB implementation crashed, we surmise that IntegriDB would be worse than TAP before the table size reaches 10 000 rows.

6.5 Scalability

In this section, we investigate the performance of TAP for realistic numbers of user IDs on a medium-end AWS machine (*m5.4xlarge*). Figure 10a shows the cost of building TAP's data structure for different numbers of IDs and sum trees. We observe a negligible difference between trees with 10 or 100 subtrees, but a noticeable difference between 100 and 1000 subtrees. In particular, building the tree takes roughly 50% more time for 1000 subtrees than for 100 subtrees: the reason is that the construction of each subtree relies on a SQL select query to obtain the leaf values (i.e., V_s), which becomes a bottleneck when the number of subtrees is large.

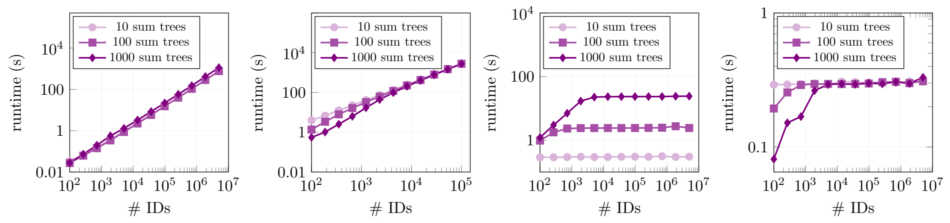
In Figure 10b, we display the cost of a full audit as a function of the number of user IDs. We observe that for large numbers of IDs, the audit cost resembles a linear function of the number of IDs. For around 15 000 IDs, a full audit takes 420 seconds, regardless of the number of subtrees.

Figure 10c displays the cost of a quantile query over the entire dataset. We see that the cost gradually becomes dependent only on the number of subtrees, as the main workload consists of creating zk-range proofs for each subtree. In Figure 10d, we display the cost of querying a fixed range consisting of 10 subtrees. The cost of the query is independent of the total number of subtrees, and only has a logarithmic dependence on the number of IDs, which is invisible at realistic scales.



(a) Server storage cost. (b) Audit cost. (c) Insert query. (d) Look-up query. (e) Sum query. (f) Min/Max query.

Figure 9: Comparison between TAP, Merkle^2, and IntegriDB in terms of storage, audit, and query costs.



(a) Tree construction. (b) Audit. (c) Quantile (full). (d) Quantile (part).

Figure 10: Performance of TAP for large numbers of user IDs.

7 Discussion & Limitations

Scaling Limitations. From Figure 10a, we observe that it takes around 285 seconds to generate TAP's data structure for ≈1.8 million IDs and 100 subtrees. This is a reasonable overhead for the smart grid use case based on SP Group, which has around 1.6 million user IDs and 84 subtrees: if one tree has to be generated per hour, then even on an \$1/hour *m5.4xlarge* machine this only takes roughly 8% of the available time. For the congestion pricing and digital advertising use cases, TAP would not be able to support a system where every possible vehicle/website participates. However, for 250 000 vehicles and 20 gantries, or for 500 websites and 10 000 advertisers, the total number of user IDs would be 5 million, which a *m5.4xlarge* machine could process in 700 seconds.

With the same machine, an auditor can audit slightly over 100 user IDs per second. As such, the upper bound for feasibility for a full audit would be 360 000 IDs for a system with 1-hour epochs. However, a randomized audit, in which a randomly chosen subset is audited, can be used for large systems to ensure that repeated misbehavior by the server is detected with high probability. Furthermore, targeted audits can be performed in cases where the operator profited from suspicious query results.

Reporting Misbehavior. The transparency property of TAP ensures that if the server misbehaves in a verifiable way (e.g., signing a demonstrably false query result), then this can be detected and disseminated through gossip or the bulletin board. Cases in which the server misbehaves by falsifying user data are harder to detect and prove. In this case, we envisage that the user would appeal to a regulator or consumer watchdog. We assume that the server has a reputation to protect, which would be tarnished by frequent complaints.

Incentives. TAP expects each user to independently and continuously verify her data and act as a whistleblower in the case of falsified data. As such, users in TAP must have an incentive to monitor measurements. In the use cases of Section 2.3, the users have a direct financial incentive to monitor their data because their bills or payments are directly impacted. From the perspective of the operator, transparency increases users' trust in the system and may help avoid frivolous lawsuits by being able to easily prove honest behavior.

Trust Model. In practice, a TAP user would not just rely on the operator for inserting data, but also for developing the app/client that verifies query proofs. In this setting, a malicious server could both falsify data *and* deliberately insert bugs into the client to falsely convince users of the validity of the proofs. It is therefore vital that TAP users who require additional security have the ability to write their own code for proof verifications.

User Identities. TAP is impractical in settings where users are fully anonymous because such a setting would allow the server to create an unlimited number of fake users. However, we assume that a regulator would be able to detect the existence of (large numbers of) fake users, especially if they have a major impact on query results (e.g., a fake website that consistently achieves the highest click-through rate).

8 Conclusions

We have presented TAP, a multi-user data service that provides data privacy, integrity, and transparency for user queries.

The ADS in TAP combines a chronological prefix tree with sorted sum trees whose roots are stored in the prefix tree leaves. This data structure allows TAP to support a wide range of queries that are useful for emerging data services, with good performance at scale.

For future work, we plan to make the privacy costs to users explicit – i.e., if an aggregate is taken that includes a sum tree with very few leaves, then this reveals more information than if the sum trees each have many leaves. This way, we can assign to each user a privacy budget [7]. Another interesting direction for future work is to add support for additional queries to TAP, e.g., Spearman rank correlation between a recent set of measurements and another set of aggregates.

References

[1] Bulletproofs implementation. https://github.com/ing-bank/zkrp.

[2] ERP gantries. https://www.sgcarmart.com/news/carpark_index.php?LOC=all&TYP=erp.

[3] IntegriDB implementation. https://github.com/integridb/Code.

[4] Merkle^2 implementation. https://github.com/ucbrise/MerkleSquare.

[5] Sixads. https://sixads.net/.

[6] SP Group. https://www.spgroup.com.sg/about-us/corporate-profile.

[7] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. Deep learning with differential privacy. In *ACM CCS*, pages 308–318, 2016.

[8] M. A. Bashir, S. Arshad, W. Robertson, and C. Wilson. Tracing information flows between ad exchanges using retargeted ads. In *USENIX Security*, pages 481–496, 2016.

[9] D. Brown. Standards for efficient cryptography 2 (SEC 2). Technical report, Certicom, 2010.

[10] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *IEEE S&P*, pages 315–334, 2018.

[11] J. Camenisch, R. Chaabouni, and a. shelat. Efficient protocols for set membership and range proofs. In *ASIACRYPT*. Springer, 2008.

[12] H. Chen, X. Ma, W. Hsu, N. Li, and Q. Wang. Access control friendly query verification for outsourced data publishing. In *ESORICS*, pages 177–191, 2008.

[13] G. G. Dagher, B. Bünz, J. Bonneau, J. Clark, and D. Boneh. Provisions: Privacy-preserving proofs of solvency for Bitcoin exchanges. In *ACM CCS*, pages 720–731, 2015.

[14] C. Dwork. Differential privacy: A survey of results. In *TAMC*, pages 1–19. Springer, 2008.

[15] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*. Springer, 2006.

[16] A. Eijdenberg, B. Laurie, and A. Cutter. Verifiable data structures. *Google Research, Tech. Rep*, 2015.

[17] N. Holohan, S. Antonatos, S. Braghin, and P. Mac Aonghusa. The bounded Laplace mechanism in differential privacy. *Journal of Privacy and Confidentiality*, 10(1), 2020.

[18] Y. Hu, K. Hooshmand, H. Kalidhindi, S. J. Yang, and R. A. Popa. Merkle^2: A low-latency transparency log system. In *IEEE S&P*, pages 285–303, 2021.

[19] Y. Ji and K. Chalkias. Generalized proof of liabilities. *Cryptology ePrint Archive*, 2021.

[20] B. Laurie. Certificate transparency. *Communications of the ACM*, 57(10):40–46, 2014.

[21] B. Laurie and E. Kasper. Revocation transparency. *Google Research*, 2012.

[22] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Authenticated index structures for aggregation queries. *ACM TISSEC*, 13(4):1–35, 2010.

[23] F. Liu. Generalized Gaussian mechanism for differential privacy. *IEEE Transactions on Knowledge and Data Engineering*, 31(4):747–756, 2018.

[24] F. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *ACM SIGMOD*, pages 19–30, 2009.

[25] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman. CONIKS: Bringing key transparency to end users. In *USENIX Security*, pages 383–398, 2015.

[26] E. Morais, T. Koens, C. van Wijk, and A. Koren. A survey on zero knowledge range proofs and applications. *SN Applied Sciences*, 1(8):946, 2019.

[27] H. Pang, A. Jain, K. Ramamritham, and K.-L. Tan. Verifying completeness of relational query results in data publishing. In *ACM SIGMOD*, pages 407–418, 2005.

[28] Y. Peng, M. Du, F. Li, R. Cheng, and D. Song. FalconDB: Blockchain-based collaborative database. In *ACM SIGMOD*, pages 637–652, 2020.

[29] R. Poddar, T. Boelter, and R. A. Popa. Arx: A strongly encrypted database system. *Proceedings of the VLDB endowment*, 12:1664–1678, 2019.

[30] D. Reijsbergen, Z. Yang, A. Maw, T. T. A. Dinh, and J. Zhou. Transparent electricity pricing with privacy. In *ESORICS*, pages 439–460, 2021.

[31] M. D. Ryan. Enhanced certificate transparency and end-to-end encrypted mail. In *NDSS*, pages 1–14, 2014.

[32] E. Shi, T. H. Chan, E. Rieffel, R. Chow, and D. Song. Privacy-preserving aggregation of time-series data. In *NDSS*, pages 1–17, 2011.

[33] R. Tamassia. Authenticated data structures. In *European Symposium on Algorithms*, pages 2–5. Springer, 2003.

[34] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *IEEE S&P*, pages 863–880, 2017.

[35] Y. Zhang, J. Katz, and C. Papamanthou. IntegriDB: Verifiable SQL for outsourced databases. In *ACM CCS*, pages 1480–1491, 2015.

[36] Úlfar Erlingsson, V. Pihur, and A. Korolova. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *ACM CCS*, pages 1054–1067, 2014.

A Pseudocode

This contains contains the pseudocode for three algorithms discussed in Section 4. Algorithm 1 determines a set of internal nodes in the prefix tree that covers the sum tree roots in the specified range, and is described under 'Range Cover'. The range cover set acts as a proof of completeness for the given range, i.e., it demonstrates that no entries were incorrectly omitted or added by the server. It relies on another function, RangeOverlap(S^*, i), which for a given interval node i in the prefix corresponds to a bit prefix that overlaps with the set S^*. (See also the function DoesPrefixOverlapRange in trees/prefix_tree.go in the GitHub repository.)

Algorithm 2 corresponds to the code that the server executes in response to a sum query, and is described under 'Sum/Count/Average/Standard Deviation' in Section 4.3. Algorithm 3 corresponds to the code that the server executes in response to a min query, and in described under 'Min/Max' in Section 4.3. The algorithm for quantiles is conceptually similar, and creates at most 2 range proofs for every sum tree.

Algorithm 1: Range proof (ExtendRangeProof)

```
Input: node $i$, node set $N$, range $S^*$
Output: node set $N'$
if RangeOverlap($S^*, i$) then
    $N' = N \cup n$
    $L = \text{LEFT}(i)$
    $R = \text{RIGHT}(i)$
    if $L \neq 0$ then
        $N' = N' \cup \text{ExtendRangeProof}(L, N, S^*)$
    if $R \neq 0$ then
        $N' = N' \cup \text{ExtendRangeProof}(R, N, S^*)$
    return $N'$
else
    return $\emptyset$
```

Algorithm 2: Sum/Count query (Server)

```
Input: range $S^*$
Output: proof $\pi$
$R \leftarrow$ PrefixTree.GetRoot()
$N' \leftarrow \text{ExtendRangeProof}(R, \emptyset, S^*)$
$L' \leftarrow \text{GetLeaves}(N')$
$v \leftarrow \text{SQL.SumQuery}(S^*)$
for $j \in \{1, \ldots, z\}$ do
    $v_j^* \leftarrow v^j$
$r^* \leftarrow 0$
for $i \in L'$ do
    $r^* \leftarrow r^* + \text{GetTotalSeed}(i)$
    $h_i' \leftarrow \text{GetRootChildrenHash}(i)$
    $c_{i1}, \ldots, c_{iz} \leftarrow \text{GetRootCommitments}(i)$
    $l_i \leftarrow \text{GetRootNumLeaves}(i)$
return $(N', L', r^*, v_1^*, \ldots, v_z^*, (h_i', c_{i1}, \ldots, c_{iz}, l_i)_{i \in L'})$
```

B Security Model

Let κ denote the security parameter, and $\emptyset$ the empty string. When X is a set, $x \xleftarrow{\$} X$ denotes the action of sampling an element uniformly at random from X.

Syntax. We define a transparent and privacy-preserving data services (TAP) scheme using the following algorithms:

$(\mathrm{P_{tap}}, k_s, \Delta_0) \leftarrow \mathsf{Initialize}(1^\kappa)$: run by the server. It takes as input the security parameter κ, and outputs system parameters $\mathrm{P_{tap}}$, a secret key $k_s \in \mathcal{SK}_{\text{tap}}$, and a mutable public verification state $\Delta_0 \in \mathcal{VT}_{\text{tap}}$ of the TAP instance, where $\mathcal{SK}_{\text{tap}}$ is a secret key space and $\mathcal{VT}_{\text{tap}}$ is the space for public verification state.

$r_{it} \leftarrow \mathsf{EpochSecretGen}(\mathrm{P_{tap}}, k_s, i, t)$: run by the server. It takes as input the system parameters $\mathrm{P_{tap}}$, the secret key k_s, and the epoch t, and outputs epoch secrets $r_{it} \in \mathcal{SS}_{\text{tap}}$ for user i, where $\mathcal{SS}_{\text{tap}}$ is an epoch secret space.

$(\Delta_t, \mathrm{R}_{it}, \pi_{it}) \leftarrow \mathsf{Query}(\mathrm{P_{tap}}, k_s, i, t, \mathsf{QType}, \Delta_{t-1}, \mathrm{M}_{it})$: run by server. It takes as input the system parameters $\mathrm{P_{tap}}$, secret key k_s, a query type $\mathsf{QType} \in \{\mathsf{insert}, \mathsf{lookup}, \mathsf{sum}, \mathsf{count}, \mathsf{average}, \mathsf{min\text{-}max}, \mathsf{quantile}\}$ from a user i for epoch t, the public verification state Δ_{t-1}, and a query message $\mathrm{M}_{it} \in \mathcal{M}_{\text{tap}}$, and outputs a query result $\mathrm{R}_{it} \in \mathcal{R}_{\text{tap}}$ and the corresponding proof $\pi_{it} \in \mathcal{PF}_{\text{tap}}$, and

Algorithm 3: Min query (Server)

```
Input: range $S^*$, large integer $K$
Output: proof $\pi$
$R \leftarrow$ PrefixTree.GetRoot()
$N' \leftarrow$ ExtendRangeProof($R, \emptyset, S^*$)
$L' \leftarrow$ GetLeaves($N'$)
$v^* \leftarrow$ SQL.MinQuery($S^*$)
$P_{zk} \leftarrow$ NIZK.SETUP($1^\kappa$)
$i^* \leftarrow -1$
for $i \in L'$ do
    $j^* \leftarrow$ GetLeftmostLeaf($i$)
    $v \leftarrow$ GetValue($i^*$)
    $c_{i1}, \ldots, c_{iz} \leftarrow$ GetLeafCommitments($j^*$)
    $h'_i \leftarrow$ GetLeafUserTimeHash($j^*$)
    $\pi'_i \leftarrow$ GenerateInclusionProof($j^*, i$)
    if $i^* = -1$ and $v = v^*$ then
        $\pi^*_i \leftarrow$ NIZK.PROVE($P_{zk}, v, [v^*, v^*+1)$)
        $i^* \leftarrow i$
    else
        $\pi^*_i \leftarrow$ NIZK.PROVE($P_{zk}, v, [v^*, K]$)
return $(N', L', v^*, i^*, (c_{i1}, \ldots, c_{iz}, h'_i, \pi'_i, \pi^*_i)_{i \in L'})$
```

an updated public verification state Δ_t (if $\mathsf{QType} = \mathsf{insert}$), where $\mathcal{PF}_{\mathsf{tap}}$ is a proof space $\mathcal{M}_{\mathsf{tap}}$ is the query message space, and $\mathcal{R}_{\mathsf{tap}}$ is a query result space.

$\{0,1\} \leftarrow \mathsf{Verify}(\mathrm{P}_{\mathsf{tap}}, r_{it}, \mathsf{QType}, \Delta_t, \mathrm{R}_{it}, \pi_{it})$: run by a user i. It takes as input the system parameters $\mathrm{P}_{\mathsf{tap}}$, epoch secret r_{it}, a query type QType, the public verification state Δ_t, and a query result R_{it} obtained from server for epoch t, and the corresponding proof π_{it}, and outputs True (1) or False (0).

$\{0,1\} \leftarrow \mathsf{EpochCheck}(\mathrm{P}_{\mathsf{tap}}, \Delta_t)$: run by auditor. It takes as input the system parameters $\mathrm{P}_{\mathsf{tap}}$ and the public verification state Δ_t for epoch t, and outputs True (1) if the verification state Δ_t is valid, and False (0) otherwise.

Given $(\mathrm{P}_{\mathsf{tap}}, k_s, \Delta_0) := \mathsf{Initialize}(1^\kappa)$, any user i's epoch secret r_{it}, any valid query message $\mathrm{M}_{it} \in \mathcal{M}_{\mathsf{tap}}$ for a time epoch t, and $(\Delta_t, \mathrm{R}_{it}, \pi_{it}) := \mathsf{Query}(\mathrm{P}_{\mathsf{tap}}, k_s, i, t, \mathsf{QType}, \Delta_{t-1}, \mathrm{M}_{it})$, we say that a TAP scheme is correct if $\mathsf{Verify}(\mathrm{P}_{\mathsf{tap}}, r_{it}, \mathsf{QType}, \Delta_t, \mathrm{R}_{it}, \pi_{it}) = 1$ and $\mathsf{EpochCheck}(\mathrm{P}_{\mathsf{tap}}, \Delta_t) = 1$.

Security Properties. Here, we define three properties of TAP: *integrity*, *transparency*, and *privacy* (to achieve the requirements R2, R3a, and R3b). In Table 5, we formulate these properties via two games $\mathsf{Game}^{\mathsf{Trans}}_{\mathsf{TAP},\mathcal{A}}$ and $\mathsf{Game}^{\mathsf{Priv}}_{\mathsf{TAP},\mathcal{A}}$ running between a challenger and an adversary $\mathcal{A}$, respectively. We model both integrity and transparency of TAP in one game $\mathsf{Game}^{\mathsf{Trans}}_{\mathsf{TAP},\mathcal{A}}$ for simplicity, since they share most of the procedures and winning conditions in the game.

In both games, the adversary is allowed to ask an oracle query $\mathcal{O}_{\mathsf{MQ}}(i, t, \mathsf{QType}, \mathrm{M}_{it})$ to query any message M_{it} of her own choice for any query type QType. Via this query, the adversary can either insert a malicious message into the data structure and also learn values based on compromised epoch secrets. Meanwhile, the epoch secrets can be compromised based on the oracle query $\mathcal{O}_{\mathsf{MQ}}(i, t, \mathsf{QType}, \mathrm{M}_{it})$. In addition, $\mathcal{A}$ may ask the oracle query $\mathcal{O}_{\mathsf{IH}}(i, t)$ which is used to insert honest messages into the target TAP instance. By doing so, $\mathcal{A}$ will try to break the transparency properties for some honest inserted messages. We model those security properties in a multiparty setting (to cover the requirement R1) because $\mathcal{A}$ can ask those queries with an arbitrary user identity.

In $\mathsf{Game}^{\mathsf{Trans}}_{\mathsf{TAP},\mathcal{A}}$, the goal of $\mathcal{A}$ is to generate a malicious message and the corresponding query results for an honest user i^* and epoch t^* (i.e., $(i^*, t^*, \widetilde{\mathsf{QType}^*}, \widetilde{\Delta_{t^*}}, \widetilde{\mathrm{R}_{t^*}}, \widetilde{\pi_{t^*}}, \widetilde{\mathrm{M}_{i^*t^*}})$) that are not generated by the challenger during the game but can pass the verification of either the honest user i^* or the auditor. To model privacy, the game $\mathsf{Game}^{\mathsf{Priv}}_{\mathsf{TAP},\mathcal{A}}$ is defined based on indistinguishability. Since the Min/Max and the quantile queries would leak the concrete value of some user (without knowing its identity), we model the privacy by letting the adversary distinguish the owner of a malicious value (chosen by the adversary) from two honest parties for all kinds of queries. This approach can also cover the privacy of the (unleaked) value of a specific user as well. After all, if the adversary can know the value of a given honest user then she must be able to break the privacy formulated by $\mathsf{Game}^{\mathsf{Priv}}_{\mathsf{TAP},\mathcal{A}}$ (i.e., distinguish the owner of the value). This leads to the following definition.

Definition 1. A transparent and privacy-preserving data services scheme TAP is secure if the advantages $\mathsf{Adv}^{\mathsf{Trans}}_{\mathsf{TAP},\mathcal{A}}(\kappa) = \Pr[\mathsf{Game}^{\mathsf{Trans}}_{\mathsf{TAP},\mathcal{A}}(\kappa) = 1]$ and $\mathsf{Adv}^{\mathsf{Priv}}_{\mathsf{TAP},\mathcal{A}}(\kappa) = \left|\Pr[\mathsf{Game}^{\mathsf{Priv}}_{\mathsf{TAP},\mathcal{A}}(\kappa) = 1] - 1/2\right|$ of any PPT adversaries $\mathcal{A}$ in the corresponding games are negligible.

Given $\mathsf{Game}^{\mathsf{Priv}}_{\mathsf{TAP},\mathcal{A}}$, the proof that TAP as discussed in Section 4 satisfies the properties of integrity, transparency, and integrity is similar to the proofs of Theorems 1–4 in [30]. Informally, Theorem 1 can be proven using the property that an adversary cannot distinguish commitments from random values, and that a hidden value can be obtained from the (known) sum and the coalition's f known values only if $n - f = 1$. Theorem 2 follows from the knowledge that a quantile query reveals at most one value per query.

C Differential Privacy

In Section 4, we have presented TAP, a data service architecture in which the server returns accurate responses to user queries at the cost of revealing a limited number of user values. However, a stronger notion of privacy may be required in some contexts. For example, if the Value attribute corresponds to power usage, and a single residence is known to be the biggest power consumer in its neighborhood with high probability, then an adversarial user can learn this residence's exact power usage through a max query on this neighborhood. *Differential privacy* [15, 32] provides a stronger notion of privacy for a data service. In differential privacy, data is obfuscated through the addition of random *noise*. We focus on

```
$\underline{\mathsf{Game}^{\mathsf{Trans}}_{\mathsf{TAP},\mathcal{A}}(\kappa)}$
$\mathsf{QL} := \emptyset;$
$(\mathrm{P}_{\mathrm{tap}}, k_s, \Delta_0) \leftarrow \mathsf{Initialize}(1^\kappa)$
$(i^*, t^*, \widetilde{\mathsf{QType}^*}, \widetilde{\Delta_{t^*}}, \widetilde{\mathrm{R}_{t^*}}, \widetilde{\pi_{t^*}}, \widetilde{\mathrm{M}_{i^*t^*}}) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{MQ}}(\cdot,\cdot,\cdot,\cdot),\mathcal{O}_{\mathsf{IH}}(\cdot,\cdot)}(\mathrm{P}_{\mathrm{tap}}, k_s)$
$r_{it} \leftarrow \mathsf{EpochSecretGen}(\mathrm{P}_{\mathrm{tap}}, k_s, i^*, t^*)$
Return $\Big((i^*, t^*, \widetilde{\mathsf{QType}^*}, \widetilde{\Delta_{t^*}}, \widetilde{\mathrm{R}_{t^*}}, \widetilde{\pi_{t^*}}, \widetilde{\mathrm{M}_{i^*t^*}}) \notin (\mathsf{QL} \cup \mathsf{HL})\Big)$
    $\wedge\ ((i^*, t^*) \in \mathsf{HL}) \wedge \Big(\mathsf{EpochCheck}(\mathrm{P}_{\mathrm{tap}}, \widetilde{\Delta_{t^*}})\Big)$
    $\wedge\ \Big(\mathsf{Verify}(\mathrm{P}_{\mathrm{tap}}, r_{i^*t^*}, \mathsf{QType}^*, \Delta_{t^*}, \widetilde{\mathrm{R}_{i^*t^*}}, \widetilde{\pi_{i^*t^*}}\Big)$

$\underline{\mathcal{O}_{\mathsf{MQ}}(i, t, \mathsf{QType}, \mathrm{M}_{it})}$:
$(\Delta_t, \mathrm{R}_{it}, \pi_{it}) \leftarrow \mathsf{Query}(\mathrm{P}_{\mathrm{tap}}, k_s, i, t, \mathsf{QType}, \Delta_{t-1}, \mathrm{M}_{it})$
$(i, t, \mathsf{QType}, \Delta_t, \mathrm{R}_{it}, \pi_{it}, \mathrm{M}_{it}) \rightarrow \mathsf{QL}$
Return $\Delta_t, \mathrm{R}_{it}, \pi_{it}$

$\underline{\mathcal{O}_{\mathsf{IH}}(i, t)}$:
$\mathrm{M}_{it} \xleftarrow{\$} \mathcal{M}_{\mathrm{tap}}$
$(\Delta_t, \mathrm{R}_{it}, \pi_{it}) \leftarrow \mathsf{Query}(\mathrm{P}_{\mathrm{tap}}, k_s, i, t, \mathsf{insert}, \Delta_{t-1}, \mathrm{M}_{it})$
$(i, t, \mathsf{insert}, \Delta_t, \mathrm{R}_{it}, \pi_{it}, \mathrm{M}_{it}) \rightarrow \mathsf{HL}$
Return $\Delta_t, \mathrm{R}_{it}, \pi_{it}$
```

```
$\underline{\mathsf{Game}^{\mathsf{Priv}}_{\mathsf{TAP},\mathcal{A}}(\kappa, \mathsf{PT})}$
$\mathsf{QL} := \emptyset;\ \mathsf{CL} := \emptyset$
$(\mathrm{P}_{\mathrm{tap}}, k_s, \Delta_0) := \mathsf{Initialize}(1^\kappa)$
$(state, i^*, j^*, t^*, \mathrm{M}_0^*, \mathrm{M}_1^*) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{MQ}}(\cdot,\cdot,\cdot,\cdot),\mathcal{O}_{\mathsf{IH}}(\cdot,\cdot),\mathcal{O}_{\mathsf{RV}}(\cdot,\cdot)}(\mathrm{P}_{\mathrm{tap}})$
    , s.t. $t^* - 1$ is the latest epoch, and all query messages have the same size,
$b \xleftarrow{\$} \{0, 1\}$
If $b = 0$, then $\mathrm{M}_{i^*t^*} := \mathrm{M}_0^*$ and $\mathrm{M}_{j^*t^*+1} := \mathrm{M}_1^*$
Else $\mathrm{M}_{i^*t^*} := \mathrm{M}_1^*$ and $\mathrm{M}_{j^*t^*+1} := \mathrm{M}_0^*$
$(\Delta_{t^*}, \mathrm{R}_{i^*t^*}, \pi_{i^*t^*}) \leftarrow \mathsf{Query}(\mathrm{P}_{\mathrm{tap}}, k_s, i^*, t^*, \mathsf{insert}, \Delta_{t^*-1}, \mathrm{M}_{i^*t^*})$
$(\Delta_{t^*+1}, \mathrm{R}_{j^*t^*+1}, \pi_{j^*t^*+1}) \leftarrow \mathsf{Query}(\mathrm{P}_{\mathrm{tap}}, k_s, j^*, t^*, \mathsf{insert}, \Delta_{t^*}, \mathrm{M}_{j^*t^*+1})$
$b' \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{MQ}}(\cdot,\cdot,\cdot,\cdot),\mathcal{O}_{\mathsf{IH}}(\cdot,\cdot),\mathcal{O}_{\mathsf{RV}}(\cdot,\cdot)}(\mathrm{P}_{\mathrm{tap}}, state, \Delta_{t^*}, \Delta_{t^*+1})$
Return $(b = b') \wedge ((i^*, t^*) \notin \mathsf{CL}) \wedge (j^*, t^* + 1) \notin \mathsf{CL}$

$\underline{\mathcal{O}_{\mathsf{RV}}(i, t)}$:
$r_{it} \leftarrow \mathsf{EpochSecretGen}(\mathrm{P}_{\mathrm{tap}}, k_s, i, t)$
$(i, t, r_{it}) \rightarrow \mathsf{CL}$
Return $r_{it}$
```

Table 5: Security games of TAP.

an approach in which the server adds noise to query results before returning them to the user [24]. An alternative design would be for the users to add random noise to their data according to a fixed probability distribution, which would allow servers to compute aggregates without learning the data of individual users [36]. However, the latter approach would prevent the server from obtaining data that may be necessary for, e.g., billing, so we focus on the former.

Improving privacy through the addition of noise necessarily reduces transparency: the noise values must be hidden (or else the obfuscated data can be reconstructed), so it is impossible to verify if they were generated in accordance with the agreed probability distributions. As such, we focus on a weaker notion of transparency, namely that a malicious server is unable to *arbitrarily* distort query results by manipulating the noise generation process. As is common in the differential privacy literature, we focus on making the magnitude of the noise dependent on the *sensitivity*, i.e., the maximum possible impact on a query result by removing a single user's value. We can then utilize zero-knowledge range proofs to prove that the noise is within some range defined by the sensitivity.

Let D be the dataset, i.e., the set of values covered by the query's range. Let $R^*(D)$ be the query's *true* result, and $R(D)$ the result that is returned by the server. Let $\mathcal{R}$ be the result space, so that $R(D), R^*(D) \in \mathcal{R}$. In our setting, $\mathcal{R}$ is the space on which our commitments and zero-knowledge range proofs are defined. Let $\mathcal{D}$ be the set of *pairs* of datasets such that for any $(D, D') \in \mathcal{D}$ a unique value $d \in D$ exists such that $D/\{d\} = D'$, or $d \in D'$ such that $D'/\{d\} = D$ – i.e., those pairs that differ in exactly one value. In this setting, the protocol satisfies (ε, δ)-differential privacy if, for all $(D, D') \in \mathcal{D}$ and all $S \subset \mathcal{R}$,

$$\mathbb{P}(R(D) \in S) \leq e^{\varepsilon} \cdot \mathbb{P}(R(D') \in S) + \delta$$

If $\delta = 0$, then the protocol satisfies "pure" ε-differential privacy, whereas if $\delta > 0$ it satisfies"approximate" differential privacy. Let the *sensitivity* of the query result R be defined as $\Delta = \max_{(D,D') \in \mathcal{D}} |R^*(D) - R^*(D')|$. A general result for $\mathcal{R} = \mathbb{R}$ [15] states that returning a result R such that $R(D) = R^*(D) + Z$, where Z is a Laplace-distributed random variable with scale parameter σ, guarantees ε'-differential privacy with $\varepsilon' = \Delta/\sigma$. In particular, $\sigma = \Delta$ guarantees ε-differential privacy.

A challenge in our context is that the Laplace distribution has positive density on the entire interval $[-\infty, \infty]$. This is acceptable in cases where the threat model assumes that the server is always honest. However, in our case it would allow a malicious server to add *arbitrarily large* noise to the true result, and therefore convince the user to accept any value desired by the malicious server. To limit the scope for server misbehavior, noise should instead be drawn from a bounded interval. Prior results in this area for truncated Gaussian [23] and Laplace [17] noise indicate that although ε-differential privacy cannot be achieved in this case, (ε, δ)-differential privacy is possible. As such, we focus in the following on a generic approach for noise on the bounded interval $\{-b, -b+1, \ldots, b\}$ for a constant b. In particular, let $g : [0, b] \rightarrow [0, \infty)$ and $G(z) = \sum_{x=0}^{z} g(x)$ such that $g(0) + 2G(b-1) = 1$. We then define the noise $Z = R(D) - R^*(D)$ as a random variable on $\{-b, -b+1, \ldots, b\}$ with the cumulative distribution function

$$\mathbb{P}(Z \leq z) = \begin{cases} G(z+b) & \text{if } z \in [-b, 0], \\ 1 - G(b-z+1) & \text{if } z \in [0, b]. \end{cases}$$

We can then prove that for Z defined in this way, (ε, δ)-differential privacy is achieved with $\delta = G(\Delta - 1)$. Due to space limitations, we leave the proof, and the use of zero-knowledge range proofs to show that $|Z| \leq b$, as future work.