Vizard: A Metadata-hiding Data Analytic System with End-to-End Policy Controls

Chengjun Cai
City University of Hong Kong
Dongguan Research Institute

Yichen Zang
City University of Hong Kong

Cong Wang
City University of Hong Kong

Xiaohua Jia
City University of Hong Kong

Qian Wang
Wuhan University

ABSTRACT

Owner-centric control is a widely adopted method for easing owners' concerns over data abuses and motivating them to share their data out to gain collective knowledge. However, while many control enforcement techniques have been proposed, privacy threats due to the metadata leakage therein are largely neglected in existing works. Unfortunately, a sophisticated attacker can infer very sensitive information based on either owners' data control policies or their analytic task participation histories (e.g., participating in a mental illness or cancer study can reveal their health conditions). To address this problem, we introduce Vizard, a metadata-hiding analytic system that enables privacy-hardened and enforceable control for owners. Vizard is built with a tailored suite of lightweight cryptographic tools and designs that help us efficiently handle analytic queries over encrypted data streams coming in real-time (like heart rates). We propose extension designs to further enable advanced owner-centric controls (with AND, OR, NOT operators) and provide owners with release control to additionally regulate how the result should be protected before deliveries. We develop a prototype of Vizard that is interfaced with Apache Kafka, and the evaluation results demonstrate the practicality of Vizard for large-scale and metadata-hiding analytics over data streams.

CCS CONCEPTS

• **Security and privacy → Privacy-preserving protocols**.

KEYWORDS

Secure data analytics; End-to-End control; Metadata privacy

1 INTRODUCTION

In today's society, healthcare, business decisions, and government operations all heavily rely on the availability of data and advanced analytic tools for accurate decision-making. However, data is often fragmented and stored locally by individuals, and the raising concerns over data leakage and unauthorized sharing [51, 87] have made it very difficult to motivate individuals to share their sensitive data. Traditionally, once the data is shared out, it goes out of the hands of the owners, and it can be copied, traded, or abused in uncontrollable manners. In a recent Pew Research survey [64], 80% and 64% respondents have concerns about how companies and governments use their data. Most respondents feel that they have little or no control over how their data is used. To remedy this severe problem, a long line of works [39, 60, 69, 76] has been devoted to constructing privacy frameworks that allow owners to define their privacy preferences and regulate data usage. But most works above need the deployment of trusted hardwares to enforce owners' policies. For many real-world data processing systems that do not apply trusted hardwares (like Apache Kafka [54]), they would still operate in a *notice and consent* mode [19] and rely on centralized trusted authorities for policy enforcement. However, many concerning data breach and misuse incidents are due to the abuse of such trusted authorities, as reported in [75].

Apart from the lack of privacy-preserving and enforceable data analytic tools that do not rely on centralized trust, it is also worth pointing out that those data policies would enlarge the attack surface and help an attacker infer owners' sensitive data. Consider an owner Alice decides to authorize his data to an analytic task q. Even though Alice can encrypt his data for confidentiality protections, with side information that q is initiated by a psychiatrist (possibly by looking up information about q on social media or the Internet), an attacker can readily learn that Alice's data will be used by a psychiatrist and thus Alice might suffer from mental illness. The first attempt to hide such policy-related metadata is to encrypt the data policies and later adopt secure computation techniques on the server side (e.g., outsourced multi-party computation [6, 12, 62]) to privately decrypt and use the data policy. However, this approach can only preserve the confidentiality of the underlying computation process. A curious server can still determine whether an owner's data have been used for a given task by observing other metadata like *data access patterns*, and infer the same amount of sensitive information about the owner as before [25, 52].

The above metadata leakage problem relates to a line of security works that strike to preserve *oblivious data access* - hiding which data have been accessed or used for a query execution. Traditionally, we can attach oblivious RAM (ORAM) [46] with secure computation techniques [1, 34, 56] to fulfill our privacy goals for both the data and its metadata. However, most ORAM constructions existed today focus only on a *single* owner setting or would rely on trusting a proxy to maintain the encrypted RAM storage. Those ORAM constructions that support *multi-owner* settings [57, 58] would generally incur heavy computation costs, making them difficult to adopt in real deployments. We note that a recent work from Chen *et al.* [25] has vastly improved the performance, but they still require a few seconds to handle each oblivious data access request.

In this paper, we present Vizard[1], an efficient and metadata-hiding data analytic system that provides full-fledged privacy preservation and enforceable control to data owners. Vizard makes customized use of a suite of lightweight cryptographic techniques

[1]Vizard is a type of mask used in the 16th century for disguise or protection.

to meet the above goals, and proposes new designs to further accommodate the needs of practical data stream processing systems that are emerging today. According to our evaluations, the newly added metadata-hiding feature in Vizard only brings around 1.12× to 1.26× overhead compared to Zeph [19], which is a state-of-the-art privacy-enforced and real-time data stream analytic system *without metadata protections*. Overall, it would take around 4.6s in Vizard to securely handle a policy-controlled analytic query over 10k owners and a time-window of 100 data stream ciphertexts. We further integrate Vizard with Apache Kafka [54] to boost our ability to handle large-scale stream submissions, and this new architecture improves system performance up to 1.8× compared to the baseline.
Vizard's Architecture. Vizard considers a classic setting where each owner communicates with two non-colluding servers [12, 25, 27, 34, 62] to outsource their data submissions and execute the analytic tasks. Like previous works, we protect the confidentiality of owners' data and policies as long as an attacker can compromise at most one server. Here to further fulfill end-to-end controls, Vizard enables release policies for the query results given by the two servers and enforces them through a decentralized byzantine-secure committee (as inspired by a recent secure data analytic framework [66]). Finally, data consumers will contact the committee for result retrievals and obtain the policy-enforced results.

Importantly, note that Vizard aims only to hide metadata leakage during the query execution phase but not the data stream submission phase. We thus need all owners to upload their data streams at a fixed frequency to avoid other timing-related pattern leakages [40]. Fortunately, this is readily achievable in our targeted data stream setting with synchronized epochs, and each server will order owners' data submissions based on the epoch number.

1.1 Overview of Our Techniques

We now sketch the technical ideas behind the constructions of Vizard. Our first observation is that our targeted problem relates to a simpler but independent problem that studies how to compute *private subset histogram*, as inspired by Boneh *et al.* [12]. In this problem, each owner i holds a string x and the servers hold a small set Q of strings. The technique for computing private subset histogram reveals, for each string $\sigma \in Q$, how many owners hold σ without leaking each owner's secret string. Recent works [12, 15] have explored a very efficient and metadata-hiding way to solve this problem with the help of a cryptographic tool called distributed point function (DPF) [14], which is essentially a secret-shared and compressed point function that has only one non-zero output (e.g., the output equals 1 for computing subset histogram).

Intuitively, by setting the non-zero output in a pair of DPF keys as each owner's data and utilizing the string x for recording each owner's policy descriptions (e.g., which query task can use his data), we can thus achieve policy-controlled and metadata-hiding analytic from the DPF-based private subset histogram process above. Now, to aggregate all policy-matched owners' data, the servers need only to use the associated description string of an incoming query (e.g., its task name) as input to evaluate all owners' DPF keys and aggregate the outputs. Despite being feasible, this basic construction is far from practical for real deployments. In particular, we need to solve the following three challenging problems.

Challenge 1: Data Stream Support with DPF. Current DPF schemes only allow each owner to embed one specific output in a pair of DPF keys. Therefore, the basic construction above would incur continuous (and burdening) DPF key construction costs on the owner side if we directly apply it for handling data streams.
Our Solution: Observing that, unlike the data values which will change at every epoch, the data policy specified by each owner might remain unchanged for a long time, meaning that most parts of the DPF key generation process (except the embedded data) are "redundant". In view of this, Vizard starts by decoupling data values from the DPF keys. Because the data values are stored outside of the DPF keys, we can allow the two servers to self-aggregate the requested data values in advance (e.g., for a time-window covering several epochs), so that only a single data value will be used as input for each owner in subsequent analytics. Vizard stores a single pair of DPF keys for each owner (as long as an owner's data policy remains unchanged) and only uses them to secretly embed controlling values of 0 or 1 when we go through all owners' data values. Ideally, assuming there are U owners, and the data value and controlling value of owner i are d_i^* and T_i respectively, what we need to compute is thus $\sum_i^U T_i \cdot d_i^*$. This resulting value is equal to the summation of all policy-matched owners' data values, since $T_i = 1$ only if the query matches owner i's policy.

The above idea is inspired by recent DPF-based private information retrieval schemes [40, 77] that require identical copies of a *plaintext* database on the two servers. To utilize it in our setting where data confidentiality should also be protected, Vizard crafts a new two-server homomorphic stream encryption scheme (in Section 4.1) to encrypt those data values with a secret key that is jointly generated by the two servers (but is not known to either server), as inspired by Doerner *et al.* [34]. Our encryption scheme outputs encrypted (but additive) database copies on the two servers, and we can apply the same technique as above to efficiently conduct secure policy-controlled analytics over data streams.
Challenge 2: Rich Data Policy Supports. Supporting a simple policy condition, i.e., $x = \sigma$, is not enough. In reality, owners might want to construct more fine-grained policies by combining multiple conditions with various operators like AND, OR, and NOT. For example, an owner might want to authorize his data to consumers that are 1) *type*= hospitals AND 2) *region*=EU.
Our Solution. Vizard enables support for all those three essential operators. Vizard starts by following a DPF-based query framework [77] (which supports secure AND and OR queries over *plaintext* database) to modify the policy constructions of each owner, so that the controlling value (i.e., 1 or 0) can properly reflect the policy logic defined by the underlying operators. Vizard improves upon prior work [77] and reduces the computation costs when evaluating policies defined with AND and OR operators (via the use of hash digests and cuckoo hashing, as shown in Section 4.3). Our optimized designs ensure that the two servers can only evaluate a constant amount of DPF keys even when the number of policy conditions scales. In addition, Vizard further supports NOT operator for owners to conveniently rule out infamous or unwanted consumers, which is done by securely inverting the controlling values with secret shares of the value 1.

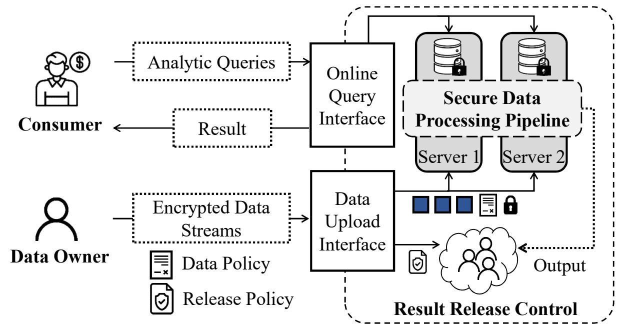


Figure 1: The architectural vision of Vizard.

Challenge 3: End-to-End Controls. In addition to enabling owners to define data policies that privately regulate how their data should be used, providing meaningful controls on how the analytic results should be released [86] is also desirable for enhancing owner protection. The challenge, however, is to enforce the result release control process without centralized trust.

Our Solution. Here in our current implementations, we have identified and provided three new types of release policies to owners, i.e., 1) *integrity-based release policy* that attests policy enforcement and result integrity via correctness proofs given by the servers; 2) *privacy-based release policy* that adds an appropriate amount of noises to the result for differential privacy (DP); and 3) *payment-based release policy* that enforces monetary rewards for the owners before releasing the results. Here to avoid allocating trust in a centralized third party, Vizard follows an emerging trend and relies on a decentralized committee (with honest majority) to jointly enforce the policies above [66, 67]. Each trustee in the committee will execute policy-specific and byzantine-secure operations to independently conduct result release controls, and data consumers will obtain the results only if the majority of trustees approve their requests.

1.2 Summary of Contributions

The main contributions made in this paper are as the following:

- A new metadata-hiding and policy-controlled analytic system in the two-server setting.
- Optimized extension methods to support rich data policies (i.e., with AND, OR, and NOT operators).
- Decentralized result release control and lightweight constructions for integrity, privacy, and payment related release policies.
- Implementation and evaluation of a prototype of Vizard that is interfaced with Apache Kafka [54].

2 PROBLEM STATEMENT

This section describes Vizard's system model, threat assumptions, and goals.

2.1 System Model

As depicted in Figure 1, there are four types of logical parties in Vizard, i.e., *data owner*, *data consumer*, a *secure data processing pipeline*, and a *result release control committee (RRC)*.

- Data owners are contributors who will continuously send encrypted data streams to Vizard (e.g., heart rates from a wearable device) for collective analytics. Along with their data, data owners can specify data policy to regulate the data usage and define release policy that jointly regulates the result release process.
- Data consumers are players who aim to learn collective knowledge (e.g., daily averaged heart rates in a specific region) by making queries to our system. Vizard's goal is to generate such knowledge in a policy-controlled and private manner.
- The secure data processing pipeline is a core computational component in Vizard, which securely operates on owners' data streams and generates the requested analytic results.
- The result release control committee (RRC) is formed by a set of stakeholders (e.g., owners, government agencies, or other third parties) to enforce data owners' release policies.

2.2 Threat Assumptions

Our secure data processing pipeline follows the emerging secure two-party computation threat model [12, 25, 40] (to list just a few) and assumes two service servers that will not collude with each other or any other party in Vizard (e.g., servers from two different cloud providers). But beyond the non-colluding assumptions, the two servers might try to *independently* infer or learn sensitive information about each data owner's data due to various interests. For honesty assumptions, rather than following a semi-honest setting and assuming totally correct executions from those two servers, we consider that they are rational economic players whose execution correctness could be questioned [30, 47, 72]. For example, they might be "lazy" servers that could avoid paying CPU and storage costs associated with our query process and return only partial or entirely incorrect results. For data consumers that aim to obtain useful insights over the collected data streams, they might also be interested in learning each data owner's sensitive information (like data or metadata).

Data owners are the contributors and beneficiaries (if we consider payment rewards from the consumers) in Vizard, and thus we assume that they will behave honestly to retain Vizard's reputation (so as to compete with other analytic services in the market). We will further discuss in Section 8 on how to effectively address potentially malicious data owners who would intentionally upload malformed (out-of-range) data inputs [12, 27]. For the RRC that is operated in a federated manner by a group of trustees, we assume a standard Byzantine security setting where, at any given time, at least $t = 2/3$ of the trustees are honest and not compromised by an attacker [55, 83]. Compromised trustees might deviate from the protocol and conduct arbitrary behaviors to jeopardize our enforcement on data owners' release policies.

In Vizard, we also assume a secure channel (e.g., TLS) and the existence of a public-key infrastructure (PKI) for each data owner and consumer to establish secure connections with Vizard. Data owners can further utilize existing anonymity networks (e.g., Tor [33] or a trusted VPN proxy) to hide their IP addresses and achieve better anonymity protections. Lastly, we assume there are out-of-bound secure communication channels between any two parties in our system for exchanging data secrets whenever needed.

2.3 System Goals

We aim to bring effective data analytic services to data consumers while assuring full-fledged privacy and control for data owners. In particular, we want to achieve the following system goals in Vizard:

- **Data Confidentiality.** No party except the corresponding data owners can learn the contributed raw data streams, and the data consumer who initiated the query can only learn about an aggregated view of those raw data streams.
- **Metadata Protection.** Given a query request from a data consumer, we want to hide the data access pattern during the query executions, i.e., no party can know *which data owners' data streams* have been used to generate the result.
- **Release Policy Enforcements.** Given a release policy that is jointly defined by data owners, Vizard aims to enforce that every query result is released to the data consumer only if it fulfills the release policy.
- **Data Stream Support.** Vizard aims to efficiently support data streams that are continuously generated from data owners, and enable time-window-based analytic queries for data consumers.
- **Rich Data Policies.** We also want to support a variety of flexible owner-centric data policies and ensure that they can work effectively and efficiently with our secure data processing pipeline.

3 BACKGROUND

This section introduces the background of existing techniques that we leverage to craft our solutions. Consider the following simplified setting: there are U data owners, and each owner i holds a data d_i and a policy string P_i that indicates the owner's preferences on data usage regulations. (For example, a data owner can specify that only a consumer located in the EU can use his data.) Now, a consumer can make an analytic query q (together with a description string σ, e.g., his location) to our system. Given q, we want to privately find the policy-matched owners and aggregate their data.

Distributed Point Function (DPF). DPF is an emerging and privacy-aware solution for accomplishing the above requirement. At a high level, DPF is constructed by secret sharing a point function f (which evaluates to zero only except one single secret element) into two function shares (say f_A and f_B), and each of them is given to one of the two servers. Each function share will not reveal the secret element anchored in f, but the summation of their outputs at any point is the corresponding output value of f.

More formally, let $f_{\alpha,\beta} : [N] \rightarrow \mathbb{F}$ be a point function such that $f(\alpha) = \beta$ and $f(\cdot)$ equals zero at any other points. A DPF consists of two algorithms $(\mathsf{Gen}, \mathsf{Eval})$. 1) DPF.Gen$(\alpha, \beta) \rightarrow (k_0, k_1)$: Given inputs α and β, generates two DPF keys k_0 and k_1 that define the two function shares of the point function f; 2) DPF.Eval$(b, k_b, \sigma \in [N]) \rightarrow \mathbb{F}$: Given an input string σ, outputs value of the function share (for $b \in \{0, 1\}$) indexed by σ. DPF guarantees that for any correctly constructed keys (k_0, k_1):

- DPF.Eval(k_0, σ) + DPF.Eval$(k_1, \sigma) = \beta$ only if $\sigma = \alpha$, and the output equals zero otherwise.
- Any attacker that compromises only one of the two servers can learn nothing about the secret index α or the value β embedded in the DPF key shares.

Here, we will use the latest DPF construction by Boyle *et al.* [15] as a black-box tool in our designs for its effectiveness. We can further adopt system-level optimization techniques like parallel sub-tree traversing [32] and one-way compression functions [77] to boost the evaluation process.

From DPF to Policy-Controlled Private Summations. The powerful DPF primitive above can readily help us build a private summation service that not only protects both the data values and policies, but also hides the data access metadata (i.e., which owners participate in a given query).

Specifically, consider now for the data owner $i \in \{1, \ldots, U\}$ that holds data d_i and policy P_i as we described before, owner i can generate a pair of DPF keys with string P_i as the secret index and d_i as the corresponding value, via $(k_{0i}, k_{1i}) \leftarrow$ DPF.Gen(P_i, d_i). Each data owner then gives DPF key k_{0i} to server 0 and key k_{1i} to server 1. This DPF ensures that the sum of two servers' outputs is the embedded value d_i only at the position indexed by P_i, and will equal zero at any other positions.

Therefore, to compute the summation of policy-matched owners' data values, for each query q, server b ($b \in \{0, 1\}$) first fetches the task description σ, and then evaluates all data owners' DPF values shared at position indexed by σ, via

$$\mathsf{sum}_b \leftarrow \sum_{i}^{U} \text{DPF.Eval}(k_{bi}, \sigma) \in \mathbb{F}.$$

Server b can then publish the value share sum_b to the other server. It is easy to understand that the sum of the values published by the two servers, i.e., $\mathsf{sum}_0 + \mathsf{sum}_1$, is the result we desired for query q. Thanks to DPF's inherent protections for both the value and the secret index, any attacker that can compromise at most one of the two servers can learn nothing about each owner's data submission and which owners have participated in a given query. Based on this simple yet efficient construction, we then show how we can extend it to support data streams and other enriched functionalities:

(a) In Section 4.1, we propose a refined construction to enable more effective data stream supports and show how to enable other aggregation functions (like variance, median, max, min) with owner-side encoding techniques.
(b) In Section 4.3, we show how to enable enriched data policies other than the simple exact match relation (e.g., $P = \sigma$) shown above (by supporting AND, OR, and NOT operators).
(c) In Section 5, we further enable owners to jointly define release policies that control how the result should be released and show how to enforce them with decentralized trust.

4 METADATA-HIDING ANALYTICS OVER DATA STREAMS

Here to support policy-controlled private summations over data streams (e.g., $\{d_i^0, d_i^1, \ldots, d_i^j\}$, where j represents the epoch number) based on our previous initial design, a straightforward idea is to create a pair of DPF keys for every streamed data in the pipeline, and let the two servers manually select the corresponding DPF keys for query executions. For example, given a stream of DPF keys $\{k_{bi}^{t_0}, k_{bi}^{t_1} \ldots, k_{bi}^{t_j}\}$, $b \in \{0, 1\}$ from owner i and a query q that asks for data at epoch t_m, each server b can fetch all DPF keys associated with epoch t_m and execute the DPF evaluation process accordingly.

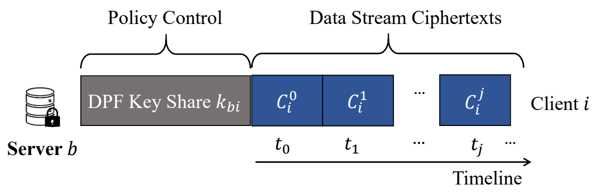


Figure 2: Our refined data storage structure (for data streams contributed by owner i) at each service server.

While being feasible, this straightforward idea would become highly inefficient if a query asks for data values in a longer time-window (e.g., $e = 100$ epochs), as the DPF evaluation costs grow linearly with the underlying time-window size (i.e., needs $e \cdot U$ times of DPF evaluations). Besides, data owners need to bear the continuous DPF key generation costs during every data submission process, which would also be highly undesirable.

4.1 Our Refined Construction for Data Streams

For the challenging issue above, our observation is that while the data stream values change over time, the data policies defined by each data owner could remain unchanged for a very long period (after an initial setup process). Therefore, the secret index of the DPF keys generated over different epochs might remain the same. Based on this important observation, our key idea is to decouple data values from the DPF keys and construct a refined storage structure for better managing each owner's data submissions (and facilitating subsequent analytic process).

As shown in Figure 2, each server will store only the latest DPF key share from every owner, and the data streams contributed by the owner are ordered by their epoch numbers and stored directly on the server. That is, the DPF keys are used now only as a secure indicator for matching the embedded data policy with analytic queries, but not for embedding the actual data stream values anymore. Here for those data stream values, we follow a highly-scalable ROM structure [34] and store encrypted copies of data stream values on the two servers (i.e., both servers store encrypted data stream values $\{C_i^0, \ldots, C_i^j\}$). Next, we describe how to construct this refined storage structure via our lightweight encryption construction below, and how to further enable efficient and policy-controlled summation queries atop this refined storage structure.

Two-server Homomorphic Stream Encryption. We build our solution by following an existing symmetric homomorphic stream encryption (SHSE) scheme [18] that can preserve data confidentiality while allowing direct operations over the stream ciphertexts.

Specifically, the SHSE scheme works as the following. Consider stream data values $\{d_0, d_1, \ldots, d_j, d_{j+1}\}$ are submitted for epoch $\{t_0, t_1, \ldots, t_j, t_{j+1}\}$ respectively, and they are integers modulo M (e.g., with size 2^{64}). Now, given a master key g^* and a secure keyed pseudo-random function (PRF) F_{g^*} that maps each epoch number t_j to a random key g_j^* in the range $[0, M-1]$, we can then encrypt each data d_j (annotated with epoch t_j) via

$$\begin{aligned} Enc(g^*, t_{j-1}, t_j, d_j) &= (t_j, t_{j-1}, d_j + F_{g^*}(t_j) - F_{g^*}(t_{j-1})) \\ &= (t_j, t_{j-1}, d_j + g_j^* - g_{j-1}^* \mod M). \end{aligned}$$

In the rest of this paper, we will deduct the epoch number t_j and t_{j-1} in the ciphertexts generated by SHSE as far as the context is clear. Note that the ciphertexts outputted from this SHSE scheme are additive via modular additions. Therefore, to compute a query q_m that covers epochs $[t_j, t_l]$, we can now significantly reduce the cost by asking each server to first *locally aggregate* each owner's corresponding encrypted data values. Also, we can decrypt the aggregated ciphertext for time-window $[t_j, t_l]$ by computing only $k_{j-1} = F_{g^*}(t_{j-1})$ and $k_l = F_{g^*}(t_l)$, as the inner keys cancel out [19].

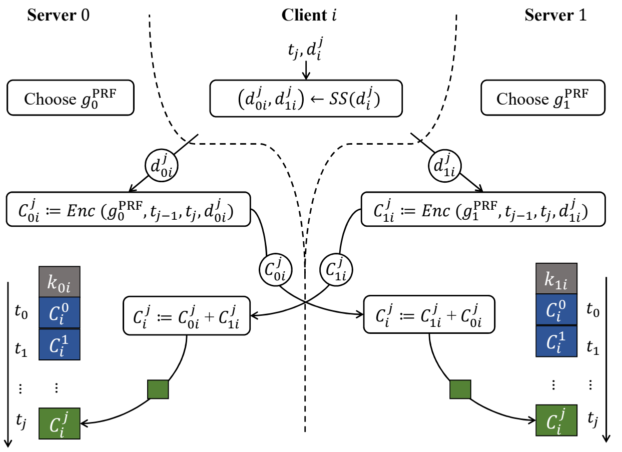

Server 0
Client i
Server 1
t_j, d_i^j
Choose g_0^{PRF}
$(d_{0i}^j, d_{1i}^j) \leftarrow SS(d_i^j)$
Choose g_1^{PRF}
d_{0i}^j
d_{1i}^j
$C_{0i}^j := Enc\ (g_0^{\mathrm{PRF}}, t_{j-1}, t_j, d_{0i}^j)$
$C_{1i}^j := Enc\ (g_1^{\mathrm{PRF}}, t_{j-1}, t_j, d_{1i}^j)$
C_{0i}^j
C_{1i}^j
k_{0i}
k_{1i}
C_i^0
C_i^1
$C_i^j := C_{0i}^j + C_{1i}^j$
$C_i^j := C_{1i}^j + C_{0i}^j$
C_i^j
t_0
t_1
t_j

Figure 3: Diagram of our two-server homomorphic stream encryption construction.

But how to use this effective SHSE scheme in a two-server setting? At first glance, it might appear that we can readily ask each data owner to generate a random PRF master key in the setup stage and use that key to encrypt the data stream values before sending them to the two servers. However, this idea would apparently cause problems when aggregating (and decrypting) over ciphertext values uploaded from different owners (as the PRF keys are different). Besides, the symmetric nature of this SHSE scheme naturally prevents us from allowing owners to jointly agree on a universal PRF master key (e.g., through a third-party provider [19]), as any compromised party can lead to disastrous privacy incidents.

With those concerns in mind, we thus propose to encrypt those values with a key jointly generated by the two servers. At a high level, inspired by the "stash-and-refresh" technique in [34], we construct our encryption process via the following two steps: 1) split data value into additive shares for each server; and 2) encrypt each share with a server's secret key and combine their outputs, as shown in Figure 3. Accordingly, we first ask each data owner to secretly share his data stream value to each server. In more detail, given a data value d, a data owner splits it into two shares d_0 and d_1 by randomly choosing $d_0 \in M$ and computing $d_1 = d - d_0 \in M$, where M is a finite field with large group size (we denote this additive secret-sharing scheme as $SS(\cdot)$). Data share d_0 is later given to server 0 and d_1 is given to server 1.

Next, to avoid the secure two-party computation (S2PC) overhead for PRF key generation and SHSE encryption, we thus perform the SHSE process independently at each server with a PRF key generated by the server itself, as shown in Protocol 1. They transmit their locally encrypted ciphertext share to the other party, and both parties can then add those shares together. The resulting ciphertext is thus the (SHSE) encryption of d_i^j (the original data stream value

Protocol 1: Two-server Homomorphic Stream Encryption. There are two servers and U data owners. Each data owner i holds a stream of data values $\{d_i^0, d_i^1, \ldots, d_i^j, \ldots\}$, and each server $b \in \{0,1\}$ holds a secret PRF key g_b^{PRF}. The protocol uses a finite field M with a large group size. For each epoch t_j, each owner i wants to encrypt his value d_i^j using the keys g_b^{PRF}, for $b \in \{0,1\}$, and replicate the result ciphertext to the two servers.

The protocol executes as the following at epoch $t_j \in I$:

(1) Each data owner $i \in \{1, \ldots, U\}$ secretly splits his prepared data value d_i^j into two shares via an additive secret-sharing scheme, i.e., $(d_{0i}^j, d_{1i}^j) \leftarrow SS(d_i^j)$. The owner sends d_{0i}^j to server 0 and d_{1i}^j to server 1.

(2) For every shared data value d_{bi}^j, each server b encrypts the value using g_b^{PRF} via the SHSE scheme:

$$\begin{aligned} C_{bi}^j &= d_{bi}^j + F_{g_b^{\text{PRF}}}(t_j) - F_{g_b^{\text{PRF}}}(t_{j-1}) \\ &= d_{bi}^j + g_j^b - g_{j-1}^b \mod M \end{aligned}$$

where $F_{g_b^{\text{PRF}}} : I \to [0, M-1]$ is a pseudo-random function.

(3) Each server $b \in \{0,1\}$ then sends the ciphertext shares $(C_{b0}^j, C_{b1}^j, \ldots, C_{bU}^j)$ to the other server.

(4) Finally, for each owner i, the two servers can obtain the result ciphertext C_i^j by computing $C_{0i}^j + C_{1i}^j \mod M$.

prepared by client i for epoch t_j) using two keys g_0^{PRF} and g_1^{PRF} (from server 0 and server 1, respectively), since

$$\begin{aligned} C_i^j = C_{0i}^j + C_{1i}^j &= d_{0i}^j + d_{1i}^j + g_j^0 + g_j^1 - g_{j-1}^0 - g_{j-1}^1 \\ &= d_i^j + (g_j^0 + g_j^1) - (g_{j-1}^0 + g_{j-1}^1) \mod M. \end{aligned}$$

Therefore, it is easy to understand that our new two-server homomorphic stream encryption design can still preserve homomorphism for its ciphertexts. For decrypting the ciphertexts aggregated for a time-window $[t_j, t_l]$, we can also ask each server $b \in \{0,1\}$ to only generate $F_{g_b^{\text{PRF}}}(t_{j-1})$ and $F_{g_b^{\text{PRF}}}(t_l)$ as the decryption keys.

DPF-based Policy Control. With two identical stream ciphertext storage at the two servers, we are now ready to introduce how we can privately conduct data control with owners' DPF keys, so that only those matched ciphertexts will be used in a query. Recall that in addition to the list of stream ciphertexts, each server $b \in \{0,1\}$ is also given a DPF key k_{bi} that encodes owner i's data policy P_i, so that only when the evaluation input $\sigma = P_i$, then the sum of the two servers' DPF evaluation equals β (which is the value embedded by owner i). What we do is to ask owners to embed a control value "1" at the secret position that is indexed by each of their data policy, so that, given a policy P_i from owner i and a query described by σ, $\text{DPF.Eval}(k_{0i}, \sigma) + \text{DPF.Eval}(k_{1i}, \sigma) = 1$ only if $\sigma = P_i$.

Now, consider a query (with description string σ) that asks for the summation over data values submitted during a time-window $[t_j, t_l]$. Let $T_{bi} \leftarrow \text{DPF.Eval}(k_{bi}, \sigma)$ be the DPF evaluation output from server $b \in \{0,1\}$ for owner i, server b can 1) locally compute owner i's ciphertext for time-window $[t_j, t_l]$ via $C_i^* \leftarrow \sum_{t=j}^{l} C_i^t$, and 2) compute its summation ciphertext share as

$$\mathsf{C}_b \leftarrow \sum_i^U T_{bi} \cdot C_i^* \mod M.$$

(As we will prove later, the sum of C_0 and C_1 is the summation ciphertext we desired for the query.)

Next, we consider how to construct the associated decryption key for the summation ciphertext above. Recall that each ciphertext is encrypted by keys g_l^b and g_{j-1}^b ($b \in \{0,1\}$) from the two servers. Therefore, assume that there are u owners' data values have been matched and used in the summation, each server b should compute $u \cdot K^b \mod M$ (where $K^b = g_l^b - g_{j-1}^b \mod M$) as its decryption key. Here to preserve privacy, each server b will compute $u_b \leftarrow \sum_i^U T_{bi}$, and then use (u_b, K^b) (for $b \in \{0,1\}$) as secret inputs to a secure two-party computation (S2PC) process to calculate

$$D \leftarrow (u_0 + u_1) \cdot (K^0 + K^1) \mod M.$$

The output will then be securely split into two additive shares (i.e., D_0 for server 0 and D_1 for server 1). After obtaining the materials above, each server b can then compute $\text{out}_b = C_b - D_b \mod M$. By obtaining out_b ($b \in \{0,1\}$), we can learn the summation of the data values of policy-matched owners for a given query q (with description σ and time-window $[t_j, t_l]$).

Correctness holds since

$$\begin{aligned} \text{out}_0 + \text{out}_1 &= \sum_i^U T_{0i} \cdot C_i^* + \sum_i^U T_{1i} \cdot C_i^* - u \cdot (K^0 + K^1) \\ &= \sum_i^U C_i^* \cdot (T_{0i} + T_{1i}) + u \cdot (-K^0 - K^1) \\ &= \sum_i^U (C_i^* - K^0 - K^1)\{P_i = \sigma\} \\ &= \sum_i^U d_i^* \{P_i = \sigma\} \end{aligned}$$

where $d_i^* := \sum_{t=j}^{l} d_i^t$ is the aggregated sum of owner i's stream data values for $[t_j, t_l]$. The generated result shares can then be delivered to the corresponding data consumer to finish the query process, and the consumer needs only to conduct a local summation process to combine the result shares and obtain the final result. Here in Vizard, the generated result shares can also be securely sent to the result release control committee (RRC) for an additional release control process (which will be introduced in Section 5).

Proof Sketch. Given that each server that will not reveal its secret DPF and PRF keys to the other server, our refined private summation design ensures the privacy of data owners' submissions (i.e., both the data streams and policies) and securely hides their access (or participation) histories for any given queries.

To start with, the security proof for policies and metadata (caused by data access leakage) directly follows from the security guarantee of DPF. Observe that each owner's policy is securely embedded in the DPF keys and a query process will access all owners' keys, which ensures that each server cannot know which owners are matched and used in the analytics [12, 34]. In the meantime, as the SHSE scheme encrypts each data value with a key jointly generated by the two servers (which cannot be learned by either server), it is apparent that the security proof for data privacy protection follows from the security guarantee of SHSE, which is based on the security of the underlying PRF [18]. Moreover, each server b cannot recover each owner's (aggregated) stream values from ciphertexts $\{C^*_{i\in U}\}$ during the query process, as it cannot learn the decryption key K^{1-b} of the other server from D_b (which is randomly generated with a secure nonce known only in the S2PC process).

Complexity. Recall that the initial construction (which directly embeds data values in DPF keys) would require the owner to continuously generate DPF keys at every epoch, causing roughly $\lambda \cdot N$ bits of communication for each server (using a PRG with λ-bit keys and N as the length of the underlying point function). In contrast, our refinement design for data streams generates only a single DPF key for each data policy and incurs $|M|$ bits of communication cost for transmitting a data share to each server. After receiving owners' data submissions, each server would need U rounds of SHSE encryption costs and a constant $|M|$-bit of communication to the other server for transmitting the ciphertext share.

For executing a given query, each server can locally aggregate ciphertexts of the identified time-window (e.g., $(l-j+1)$ rounds of aggregation for $[t_j, t_l]$), U rounds of DPF evaluations and ciphertext constructions, and one round of S2PC-based multiplication process for calculating the decryption share. Here, such S2PC-based multiplication can be facilitated by owner-aided Beaver's triplet tricks [6] to avoid server interactions, although it would result in an extra (amortized) 3log$|M|$ bits sent to each server.

4.2 Aggregation Statistics beyond Summation

Vizard can support many other aggregate statistics beyond summation. Below we sketch some useful examples.

We first consider how to compute the mean of a set of policy-matched stream values. As Vizard naturally can compute the number of matched owners by calculating a private subset histogram, i.e., by asking each server b to output $u_b \leftarrow \sum_i^U T_{bi}$ and compute their sum, we can thus obtain the mean by dividing our summation result by the number of matched owners. To support analytic functions other than mean, we can readily leverage existing owner-side encoding techniques [18, 19, 27, 77] (to list a few) to map a data value to a vector with different statistics (e.g., variance) and execute element-wise additions to obtain our desired results. For example, by asking each owner i to encode his value as (d_i, d_i^2), we can thus compute the variances of a set of policy-matched data stream values by calculating $\sum_i^U (d_i^2) - \sum_i^U (d_i)^2$.

While for non-additive statistics (e.g., max, min, median, range), we can further follow a seminal work by Corrigan-Gibbs *et al.* [27] and ask each data owner i to represent his value d_i as a length-D vector of bits $(b_0, \ldots, b_{D-1})$, where D is the range of the data value (e.g., 0-200 km/h for traffic monitoring) and $b_j = 1$ if and only if $d_i = j$. Then, the secure element-wise summation of those bits can thus reveal the required max, min, median, range, and many other useful statistics. We refer readers to [27] for optimizations of data values with a large range and more advanced techniques for linear model training that can be supported in Vizard.

4.3 Supporting Rich Data Policies

In our above designs, we only allow owners to specify a DPF-based policy function that only matches one specific query description string, e.g., "region = EU". But in reality, we observe that owners might want to specify a data policy function that can utilize multiple conditions and the essential AND, OR, NOT operators. For example, an owner contributing his daily heart rates might want to authorize the usage to a consumer whose "type=hospital" AND "region=EU", so that he can authorize data usage rights to hospitals in the EU only. We start with a prior work from Wang *et al.* [77] that tries to achieve a similar goal by using multiple DPF keys.

Prior Work: Conditioned Private Query via DPFs. Wang's work has explored how to let data consumers privately define SQL-like conditions for controlling the outputs of their private queries over a *plaintext public database*. For example, they can let a data consumer privately query data values associated with a set of secret labels (e.g., age="18" and job="lawyer"). Similar to us, they achieve this by assuming identical storage on the two servers and constructing DPF keys whose summation evaluates to 1 only when the conditions are matched. By modifying this basic construction, they are able to enable two types of enriched conditions below[2]:

- AND *conditions.* For conditions of the form $c_1 = secret_1$ AND $c_2 = secret_2$ AND … AND $c_n = secret_n$, they concatenate those secret strings $secret_1||secret_2||\ldots||secret_n$ and use it as input to the DPF key generation process. Only when all associated conditions in a query are matched, then the controlling value will be equal to 1.
- OR *conditions.* For conditions of the form $c_1 = secret_1$ OR $c_2 = secret_2$ OR … OR $c_n = secret_n$, they generate n DPF keys, with each embedding a secret string. The controlling value is calculated by $\sum_{b\in\{0,1\}} \sum_{j=1}^{n} \mathsf{DPF.Eval}(k_b^j)$, and it is equal to 1 if only one of the conditions is matched.

Their condition designs above can be directly adopted in Vizard for supporting enriched data policies. However, it falls short in performance if an owner aims to define a larger set of conditions. For example, the DPF key size for the AND operator and the evaluation costs for the OR operator will grow linearly with the number of conditions. Besides, it cannot support the NOT operator, making it difficult for an owner to efficiently filter out some infamous data consumers or unwanted query tasks.

4.3.1 *Our Refined Design for Data Usage Control.*

In Vizard, we want to support those three essential operators (i.e., AND, OR, and NOT) while keeping the costs affordable even with a large set of conditions. We start with the AND operator.

AND Operators. Here to reduce the DPF key sizes when handling a large set of condition inputs, our key idea is thus to add an extra secure function for mapping owners' policy conditions to a smaller size digest. For example, we can use a cryptographic hash function H for compressing the secret input strings (e.g., $secret_1||secret_2||\ldots||secret_n$ from owner i) into a hash digest dig_i, and later generate the DPF keys using dig_i as the input. Consider that H is a publicly known function (e.g., SHA256), each of the two servers can fetch associated description strings from a query, generate a hash digest, and finally use the digest for DPF evaluations.

OR Operators. It might appear that the above idea can be generalized to OR operators. For example, given a set of conditions

[2]We note [77] can also enable range conditions, but it requires an extended DPF function for representing an interval function [14], and thus we omit the design for range conditions here for simplicity. We will explore extending DPF to other advanced function secret sharing (FSS) constructions [13] in our future works.

$c_1 = secret_1$ OR $c_2 = secret_2$ OR ... OR $c_n = secret_n$, we can generate an index digest for each input via $\mathsf{ind}_j \leftarrow H(secret_j)$, and use ind_j to label the DPF keys embedding $secret_j$. Therefore, with the labeled DPF keys from owner i, i.e., $(k_{bi}^{\mathsf{ind}_1}, ..., k_{bi}^{\mathsf{ind}_n})$ for $b \in \{0,1\}$, each server b can first hash the description string to obtain an index digest, and then find the corresponding key for subsequent DPF evaluation process. Although this design can effectively achieve $O(1)$ DPF evaluation costs, now the two servers can reveal the secret input embedded in every DPF key via off-line hash searches, which obviously violates DPF security.

Here to remedy this threat, our key idea is to always identify a fixed amount of DPF keys for the servers to evaluate, so as to hide 1) whether there is a matched DPF key and 2) which DPF key is matched. Specifically, given a description string σ and n DPF keys generated by an owner, each server will be instructed to evaluate p (with $p \leq n$) DPF keys. Those keys can consist of p dummy keys randomly selected from the n keys, or a matched key and $p-1$ dummy keys. To prevent the two servers from distinguishing these two settings, our design further follows the idea of Castro *et al.* [31] and constructs a set of random mapping functions for key selections.

Let $\mathsf{RM} : \{0,1\}^\lambda \times \{0,1\}^N \rightarrow [m]$ be a "string-to-integer" random mapping seeded by $\theta \leftarrow \{0,1\}^\lambda$, which can be constructed (given string $secret_j$) as $\mathsf{id} \leftarrow H(\theta, secret_j) \bmod (m+1)$, where H is a cryptographic hash function salted by θ and $m \geq n$. If we generate different salts $\{\theta_1, \ldots, \theta_p\}$, for each input string $secret_j$, we can thus obtain p different indexes. But instead of putting the DPF key generated for $secret_j$ in all those p indexes, an owner can randomly pick one and insert the key to that position (if all positions are filled, the cuckoo hash rule can be adopted to kick and reinsert an existing key). The positioning of those generated DPF keys will be finalized if all keys are inserted, and the rest positions will be filled with fake DPF keys (e.g., using 0 as their secret inputs). All the keys (together with the salts) will be given to the two servers. (In Appendix B, we will show how to properly choose the parameters p and m to maintain an acceptable success probability for this positioning task).

Now, for each description string input σ, each server first computes p buckets that σ could lie, i.e., $\mathsf{id}^* \leftarrow \mathsf{RM}(\sigma)$ with all p salts. Finally, each server b will evaluate the p DPF keys identified by $\{\mathsf{id}_1^*, \ldots, \mathsf{id}_p^*\}$ using σ and compute the controlling value T_{bi} (for client i) as

$$T_{bi} \leftarrow \sum_{j \in [p]} \mathsf{DPF.Eval}(k_{bi}^{\mathsf{id}_j^*}, \sigma) \bmod M.$$

NOT Operators. Achieving the support for NOT operators requires exactly the opposite of what we can get from the DPF evaluation process, i.e., output zero if the owner's policy matches, and 1 otherwise. A straightforward idea to achieve this is to craft let owners create a large amount of (popular) conditions with the OR operators. While we can achieve constant evaluation cost even though the number of DPF keys are large (thanks to our optimized design above for OR operators), it would still incur enormous costs on the owner side for generating and transmitting those DPF keys. In contrast, our key idea is to design a simple transformation to let the DPF key evaluation results output the opposite value. Specifically, for a pair of DPF keys from owner i that emulates the point function $f_{P_i,1}$ (i.e., it outputs 1 only on a secret index P_i and equals zero otherwise), what we do is to let the two servers evaluate i's controlling value T_i as $T_i \leftarrow 1 - (\mathsf{DPF.Eval}(k_{0i}, \sigma) + \mathsf{DPF.Eval}(k_{1i}, \sigma))$, so that

$$T_i = \begin{cases} 0 & \text{if } P_i = \sigma \\ 1 & \text{otherwise} \end{cases},$$

which is the results we want for NOT operators. To fulfill this idea, we can let the two servers jointly create additive shares of the value 1, e.g., via $(v_0, v_1) \leftarrow SS(1)$. In this way, each server b can compute $T_{bi} = v_b - \mathsf{DPF.Eval}(k_{bi}, \sigma) \bmod M$ for owner i. Since v_0 and v_1 are additive shares of 1, $T_{0i} + T_{1i}$ is thus the controlling value we desired for NOT operators.

Remarks. From our refined designs for AND, OR, and NOT operators above, it is clear that each type of operator requires a unique process for correct executions. Hence owners should clearly specify the underlying operator type for each generated DPF key, so that the two servers can select the proper process for query handling. Besides, instead of using each operator separately, we can also support mixed use of the operators, e.g., NOT in $(secret_1$ AND $secret_2)$, by properly combining the process of AND and NOT operators.

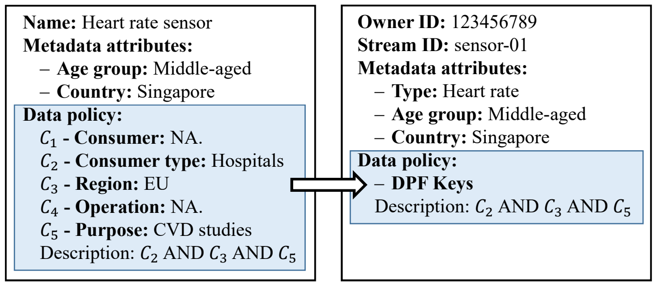
5 RELEASE POLICY ENFORCEMENTS

Recall that after processing a query, each server b will obtain a result share out_b. In this section, we will illustrate our next step that aims to enable result release policies and their enforcement during the executions. As introduced before, Vizard relies on a committee of trustees to handle the release policies. To deliver a result, the result share from each server will be securely given to each trustee via a threshold secret sharing scheme (e.g., Shamir's scheme [70]), and the result can be recovered if a majority of trustees (e.g., more than 2/3) agree to hand out their local shares.

5.1 Committee and Policy Settings

Committee Setups. The committee for release policy enforcements, denoted as RRC, is comprised of volunteering nodes from various sectors that want to join and enforce release policies. Each volunteering node will first register to our system, and then, we will periodically select a group of volunteering nodes as trustees to form RRC. Any secure node selection methods (that can ensure byzantine security) can be utilized to form RRC. Here to prevent targeting attacks (where an attacker knows which nodes will be selected and compromise them in advance) that will weaken the security of RRC, we can further adopt a verifiable random function (VRF) based approach [45, 66] for randomized node selections. After a predefined period (e.g., three months), a new batch of trustees can be selected for security.

Aggregated Release Policies. The results given from our system are collective insights contributed by data owners. Therefore, it is desirable to respect the release policies specified by each data owner. But instead of directly following each of those policies (which could be very diverse and lead to conflicting results), our idea is to narrow down the policy choices and generate an aggregated view that properly reflects all owners' policy preferences. Specifically, Vizard carefully selects and supports the following three essential types of release policies in our current implementations. They are:



Name: Heart rate sensor
Metadata attributes:
– **Age group:** Middle-aged
– **Country:** Singapore
Data policy:
C_1 **- Consumer:** NA.
C_2 **- Consumer type:** Hospitals
C_3 **- Region:** EU
C_4 **- Operation:** NA.
C_5 **- Purpose:** CVD studies
Description: C_2 AND C_3 AND C_5

Owner ID: 123456789
Stream ID: sensor-01
Metadata attributes:
– **Type:** Heart rate
– **Age group:** Middle-aged
– **Country:** Singapore
Data policy:
– **DPF Keys**
Description: C_2 AND C_3 AND C_5

Figure 4: Example of the data stream descriptions for a heart rate sensor (*left*) and our secure transformation for its data policies (*right*). Vizard preserves public metadata attributes (e.g., age group and country in this example) to facilitate grouping and filtering of different data streams.

Consumer ID: 1456164
Data requirements: [Health care data; Middle-aged; Southeast Asia]
Time window: From [2020-8-28] to [2021-10-12]
Description: [1456164; Hospitals; EU; Summation; CVD studies]

Figure 5: Example of an analytic query. In addition to query information (e.g., data requirements and time-window) that help locating the demanded data streams, a description string corresponding to options $\{C_1, \ldots, C_5\}$ will also be included.

1) *Integrity-based policies* that ensure result correctness and data policy enforcement; 2) *Privacy-based policies* that address the severe privacy leakage problem caused by the results [12, 66, 67]; and 3) *Payment-based policies* that bring fair incentives for owners and motivate active data contributions [21, 74].

For each policy choice above, we can aggregate the parameter preferences of all owners (e.g., the privacy budget on the result and the payment rates) and generate a unified release policy. Due to space limits, we only present the design ideas of each release policy above, and readers can refer to Appendix C for the details.

5.2 Integrity-based Release Policies

Recall that owners' data policies are privately embedded (in the form of DPF keys) in the secure computation process between the two servers. Therefore, in order to enforce that owners' policies are respected during the analytic process, we only need to ensure that the two servers have faithfully executed our computation protocols.

There are many existing solutions to fulfill the above goals. For example, we can leverage the lightweight "ringer" technique [47] to effectively enforce result integrity in a probabilistically-secure way (under the weaker "lazy-but-honest" server setting). In the malicious setting, we can also leverage publicly verifiable multi-party computation (verifiable MPC) techniques [30, 73] (to list just a few) to check the integrity of the results given by the two servers.

5.3 Privacy-based Release Policies

Apart from enforcing execution correctness of the two servers, endowing the aggregate statistics with differential privacy (DP) is also highly desirable. With the enforcement of such an essential policy, owners might be more willing to contribute data as their privacy is better protected against sophisticated statistical attacks [86].

Vizard follows the security framework in [66, 67] and relies on the RRC to correctly generate DP noises. Specifically, each trustee will generate a DP noise locally, and a secure computation process (initiated by the trustees or the servers) will then privately sum every trustee's noise and add the noise to the result. Achieving this requirement in the byzantine setting is non-trivial, since we need to ensure that the summation of all trustees' noises follows the privacy budget (ϵ, δ) that is jointly decided by data owners. In Vizard, we address this by adopting a byzantine-secure decentralized noise addition scheme [71] to generate the noises, so as to enforce that every result is protected with a pre-defined amount of DP noises.

5.4 Payment-based Release Policies

This policy aims to enforce that the result is revealed to the consumer only if the consumer has made the required payment. Vizard relies on the blockchain (e.g., Ethereum [80]) to form a transparent payment log which all trustees in the RRC can agree upon, so that they can later locally decide whether to help the consumer recover the result or not. This ensures that the consumer can recover the result only if it has paid, as the consumer cannot fraudulently claim that it has paid on the blockchain and convince the majority of trustees in the RRC. (To boost confidence on a payment, each trustee might need to wait for a few confirmation time, e.g., 6 blocks, before making decisions.)

6 SYSTEM IMPLEMENTATION

Vizard aims to provide full-fledged protections and rich policy supports to the data owners while allowing effective data analytic services. Here to facilitate practical deployment of Vizard in real scenarios, below we show some of our designs on the implementation sides that focus on 1) data policy preferences and query formats, and 2) our integrated data stream processing pipeline that is interfaced with Apache Kafka [54].

6.1 Data Policy and Analytic Queries

Vizard allows owner-centric data policies and respects them in a privacy-preserving manner. While there are many policy options that an owner can specify, we suggest and provide a sensible and public set of options in our current implementation for demonstration purposes. Specifically, each owner can specify his preferences for the following five options: 1) C_1: **Consumer**. This option specifies which consumer can (or can't, by using the NOT operator) compute on his data (e.g., via the public consumer IDs); 2) C_2: **Consumer types**. This option is a relaxed version of C_1, which specifies the allowed type of consumers; 3) C_3: **Region**. This option specifies the region requirement of the consumer; 4) C_4: **Operation**. This option specifies the allowed operations (e.g., summation, variance, etc.); 5) C_5: **Purpose**. This option specifies the allowed usage of his data. (Strings used to fill in each option above are standardized to facilitate the owner-consumer matching.)

The above options will be securely transformed to DPF keys together with a description that illustrates how those options will be processed on the server side. For example, as shown in Figure 4, a data owner can specify that he only allows hospitals in the European Union to do cardiovascular disease (CVD) studies over his contributed data streams. It is done by filling in associated options

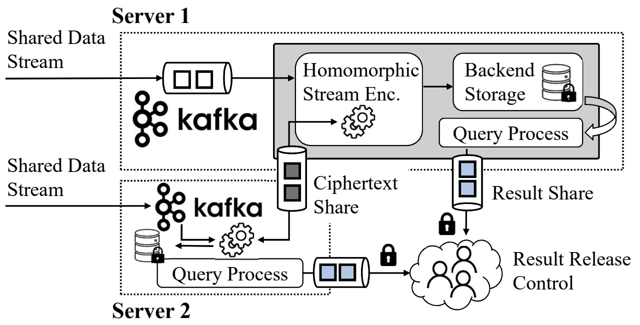


Figure 6: Architectural vision of Vizard's integration with the Kafka data stream pipeline.

(and neglecting irrelevant options), generating DPF keys according to the underlying operators (e.g., AND operator in this example), and describing their relations as "C_2 AND C_3 AND C_5".

In order to allow the two servers to correctly match query to those specified data policies, a consumer will provide a description string denoting their answers to those options ($\{C_1, \ldots, C_5\}$) in their query, and these strings will be used as inputs for DPF evaluations when processing each owner's encrypted data values. For example, "Hospital", "EU", and "CVD studies" will be fetched from a query shown in Figure 5 when processing the data policy above (i.e., "C_2 AND C_3 AND C_5"). (Note that the description string from a consumer will be authenticated before it can be used in Vizard).

6.2 Integration with Apache Kafka

Apache Kafka [54] is one of the most popular data stream processing platforms that can preserve good load-balancing and fault-tolerant protection. However, despite its ability to handle large-scale data streams, Kafka normally operates in a centralized setting with plaintext data streams. Our goal here is to integrate Vizard with Kafka, so that we can enjoy efficient processing speed while providing sufficient privacy protection to data owners.

Here in Vizard, we run an independent Kafka cluster on each server as its interface for handling data submission and retrieval requests, as in Figure 6. Specifically, Kafka will be used 1) as a data pipeline for buffering (shared) data streams submitted by data owners; 2) as a ciphertext assembly pipeline to push the generated ciphertext shares to the other server (i.e., for completing the two-server stream homomorphic encryption process); and 3) as a gateway for securely delivering result shares to each trustee in the RRC. Each party (including owners and servers) will interact with the two Kafka clusters respectively via inherent Kafka APIs, and we will enable broker replications to boost the performance.

6.3 Implementation Details

Our prototype of Vizard is implemented on top of the Spring framework [43] (which is an open and flexible framework for Java development) and Apache Kafka [54], consisting of roughly 3000 SLOC. We used Java in most of our implementations, but we also used native codes in C++ for DPF and PRF functions via the Java native interface. Specifically, we adopted the DPF implementation from Kales *et al.* [53] and used AES with 128-bit keys to implement PRF protocols (and PRG inside DPF), along with CPU-based AES-NI and

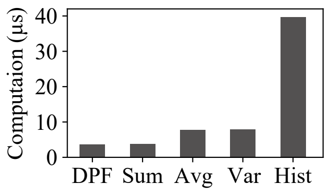


Figure 7: Owner cost for generating DPF keys and shares for different stream encodings: sum, average, variance, and histogram (with ten buckets).

other intrinsic instructions for boosting speed. We implemented a standard additive sharing scheme with a finite field of group size 2^{64}. We used an integrated framework [42] for communicating with each server's Kafka cluster. For result release control, we utilized existing cryptographic libraries for Pedersen commitment and Shamir's secret sharing, and realized a standard geometric distribution tool in Java for noise generation. We also implemented a basic version of Zeph [19] (with one privacy controller) for comparisons.

7 EVALUATIONS

We evaluate the effectiveness of Vizard by focusing on its ability to handle large-scale data streams. The experimental evaluation consists of three parts that test the performance on the data submission, data processing, and result release stages respectively.

Experiment Setup. We run the benchmarks for owners, servers, RRC trustees, and consumers on Amazon EC2 instances (m5.xlarge, 4 vCPU, 16 GiB, Ubuntu Server 20.04 LTS). Additionally, we leverage Amazon MSK [41] to enable a Kafka cluster for each server on an instance with the same configuration above, and further employ an additional Amazon RDS server (m5.xlarge with 100GB gp2 SSD) as each server's MySQL database instance for hardening system security. Servers are deployed on two EC2 instances spread over two availability zones in the Asia Pacific (Hong Kong) region, and RRC trustees are placed in different availability zones in this region also to simulate federations. This configuration helps us to benchmark in a controlled environment where network fluctuations are less likely. Based on our evaluations, the bandwidth of each instance in this environment is 5Gbps and the round trip time (RTT) between any two instances is around 0.58ms. We employ the Java microbenchmark harness tool [9] for performance evaluations.

7.1 Data Owner

Computation. We start with the computation costs for a data owner to create data stream submissions, which include the construction of DPF keys and data shares. From Figure 7, we can see that both procedures above are efficient, which need only around 3.6 μs to generate a pair of DPF keys for a data policy and around 3.7 μs for additive share generations. We also test the speed for different encodings (i.e., average, variance, and histogram), and we can observe that the throughput ranges from 25k to 130k submission per second, depending on the underlying encoding.

Bandwidth. The submission size for each server will be determined by the number of DPF keys and the number of data elements in the encoding. But overall, it only requires around 26.76 KB to simultaneously transmit 100 shared data elements and 10 DPF keys (each takes 1020 Bytes), which we think is affordable for both PC users and mobile phone users.

Table 1: Throughput (r/s) vs. Kafka stream partitions and broker replications (partition=1 is the comparison baseline). R represents the replication factor.

Partition Numbers	Submission = $1k$		Submission = $1M$	
	$R = 1$	$R = 3$	$R = 1$	$R = 3$
1	$2.85k$	$2.78k$	$301.93k$	$212.72k$
3	0.99×	0.98×	1.42×	1.16×
5	1.02×	0.99×	1.40×	1.51×
10	1.06×	1.02×	1.41×	1.84×

Table 2: Microbenchmark on each server for conducting two-server homomorphic encryption.

# Submissions	Enc.	Comm.	Add.	Database (opt.)
100	0.5 ms	68 ms	0.1 ms	0.3 s
$1k$	6.3 ms	74 ms	0.1 ms	1.9 s
$10k$	54.3 ms	118 ms	1.1 ms	12.3 s
$100k$	555.1 ms	321 ms	11.6 ms	120.5 s

7.2 Data Processing Pipeline

Throughput for Handing Submissions. Recall that after generating their submission shares, data owners will stream each share to one server for processing via its Kafka cluster. The Kafka cluster is used as a buffer for coping with large-scale data stream submissions. At its core, it comprises two main APIs: a producer API that imports data stream to the cluster and a consumer API that exports the requested streams from the cluster. From Figure 8, we can see that it can nicely scale to handle around 10 million stream submissions, and the performance bottleneck is on the front end where the Kafka cluster imports and handles submission shares from data owners.

Next, we explore how this submission handling throughput evolves when we consider submission partitions and broker replications (for distributing the load and fault-tolerant protection, respectively). Table 1 shows the results in different submission scales. From the table, we can learn that increasing both the partition and broker numbers can help boost the performance when the submission scale is large enough (e.g., 1 million submissions per second), and the trade-off between robustness and performance is also acceptable for real deployments.

Two-server Homomorphic Encryption. Each server will consume data streams from its Kafka cluster and conduct the encryption process for generating all owners' stream ciphertexts. We plot the costs for computation (i.e., local encryption and ciphertext share addition) and communication between the two servers in Table 2. From the table, we can observe that the main bottlenecks are the time costs for encryption and ciphertext share transmissions (which are conducted through monitoring the other server's Kafka cluster). But overall, it only takes less than a second for the two servers to jointly encrypt $100k$ shares. The generated stream ciphertexts could be stored in an SQL-enabled database to facilitate subsequent queries, which takes an additional and linearly growing time costs for database insertions (e.g., 12.3s for inserting $10k$ records).

Query Cost. Once an analytic query process is triggered, each server will 1) locally aggregate all specified ciphertexts (i.e., based on the time-window), 2) securely evaluate all owners' DPF keys, and

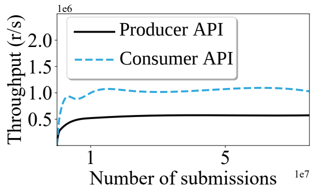


Figure 8: Kafka throughput for data stream submissions.

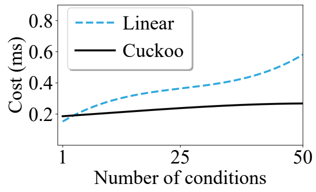


Figure 9: The DPF key evaluation cost for OR operators.

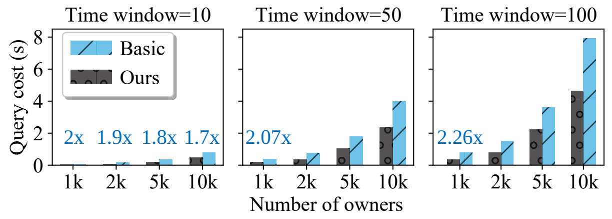


Figure 10: The time cost for conducting secure queries w.r.t. the number of owners and the time-window length.

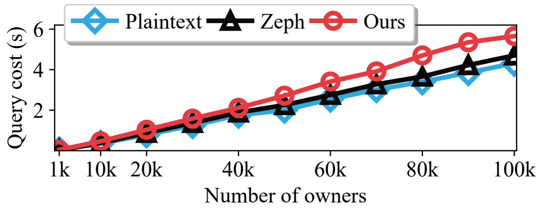


Figure 11: Query cost comparison with plaintext queries and Zeph [19] (with a time-window of 10 ciphertexts).

3) jointly decrypt and generate a share of the result. Figure 10 shows our evaluation result and its comparison with the basic solution illustrated in Section 3 (which readily embeds data in the DPF keys). It turns out that our query cost scales with the owner numbers and the time-window length (for retrieving more ciphertexts from the database). But overall, it takes only around 4.6s to query over $10k$ owners with a time-window of 100 ciphertexts, which is 1.7× less than the basic construction. We observe that such performance gain will moderately decrease when the number of owners grows (as we require additional processing steps other than DPF key evaluations), but in our evaluations it can still achieve 1.4× less time cost with $100k$ owners (and a time-window of 10 ciphertexts), and the gain will grow noticeably when we query a larger window of ciphertexts.

We further compare Vizard's query process with that in Zeph [19], which is a state-of-the-art data analytic system that enforces privacy control but does NOT provide metadata (or policy) protection, in Figure 11. The figure shows that Vizard only incurs an additional 1.12× to 1.26× latency overhead compared to Zeph, which we think is an affordable security trade-off for the query process.

Finally, we zoom in to focus on the DPF key evaluation process and evaluate the cost for supporting enriched policies (i.e., with AND, NOT, and OR operators). From the results, we find out that the execution of AND and NOT operators adds a very slight cost ($< 0.1ms$) to the evaluation process. Figure 9 further compares our optimized approach for the OR operator with the standard approach that linearly evaluates DPF keys, and the results confirm the performance gain of our approach (e.g., 2.17× less for 50 OR conditions).

7.3 Result Release Control

Lastly, we evaluate the release control cost for delivering one result with an RRC consisting of 10 to 30 trustees. Here for simplicity, we omit the initialization time cost for RRC formations and focus only on the added cost for each trustee and the two servers due to our release controls. We start with the result share splitting process on each server. In our evaluation, it requires less than $0.6ms$ to compute shares for an RRC of 30 trustees and less than $1ms$ to deliver each share to one trustee via its Kafka cluster, which is efficient to ensure a smooth delivery process.

Cost for Release Policy Executions. We evaluate the performance of Vizard's release policies with "lazy-but-honest" servers. Our integrity-based policy follows the widely adopted "ringers" [47] technique to check servers' integrity in a probabilistically-secure way. That is, we insert challenge queries and their commitment tokens (which are prepared in advance by owners) in a batch of new queries and secretly share the answer (e.g., the challenge queries' ids) to each trustee in the RRC. This technique can vastly ease the cost on the server sides for integrity proof generations (i.e., finding which queries in the given batch are the challenge queries, as shown in Appendix C.1). In our evaluation, it only adds around $2.25ms$ per query for the two servers. Once the servers output their proofs, answer recovery is required for each trustee to validate the proof. Although the recovery cost grows with the number of trustees, the overall cost remains moderate (e.g., less than 0.25s for 30 trustees).

Our decentralized DP-based release policy is output perturbation-based [86], which asks each trustee to sample byzantine-secure noise locally. We evaluate the noise addition cost with a secure computation process initiated by the two servers, i.e., each sampled noise will be treated as a data submission for the two servers and added during the query process. From Table 3, we can see that both the noise generation and addition process are at the microsecond scale, which are very efficient. Last but not least, for the payment-based policy that asks each trustee to check against a consumer's payment proof on a public ledger like Ethereum, we evaluate the proof checking cost using a popular infrastructure named BlockCypher [11]. The result shows that each trustee requires around 1.5s to complete the check, which is acceptable given that a consumer can make a batch payment for a large number of queries.

Here we omit the cost for result delivery from RRC to a consumer, as it largely depends on the location and network condition of the consumer and trustees. After collecting all required shares from the RRC trustees, our release process would bring an additional 1.7 to 272 ms computation cost for the consumer to recover the result (or DP-protected result), depending on the RRC size.

8 EXTENSIONS

In this section, we discuss some extension opportunities for Vizard.

Addressing Malicious Owners. We can provide protections against malicious owners that might inject carefully crafted inputs (e.g., out-of-range data inputs) to Vizard in order to influence the analytic outputs more than they should. Fortunately, this problem has been extensively studied in the multi-server settings since the seminal Prio system [27], and it is covered in recent efforts that aim to provide verifiability to the generated DPF keys (in either the semi-honest [15] or the fully malicious server setting [12]). Vizard can adopt those techniques to strengthen our defense against malicious clients, ensuring that 1) their data shares are within a valid range, and 2) their DPF keys indeed evaluate to a correct point function that has at most one non-zero component.

Table 3: Computation Cost for Release Control vs. RRC Size.

RRC Size	Integrity (batch=20)		Privacy		Payment
	Proof Gen.	Check	Noise Gen.	Add.	Check
10	45 ms	1.83 ms	3.9 μs	0.1 μs	~1.5s
20	45 ms	30.62 ms	3.9 μs	0.1 μs	~1.5s
30	45 ms	242.06 ms	3.9 μs	1 μs	~1.5s

Parallel Accesses. One performance setback in Vizard is the linear DPF key evaluation costs: namely, each server needs to go through all clients' keys to obtain the correct results. While we note that this is a security trade-off for providing metadata protections, we can boost the performance by letting each server evaluate every owner's DPF keys in parallel. Also, the two servers can create multiple instances, so that each pair of instances can handle a query request in parallel and boost the batch query performance.

9 RELATED WORKS

Now we give a summary of some closely related works.

Data Analytics with Enforced Privacy. Enabling data analytics while protecting the privacy of the underlying data is a well-studied topic. Throughout the years, many practical solutions, which base on 1) a centralized server and advanced cryptographic encryption schemes [3, 66, 67, 71], 2) non-colluding servers and MPC-based secure computations [27, 50, 62], or 3) trusted hardwares [35, 65, 85], are proposed. Although data confidentiality is well preserved in existing works, the importance of metadata privacy of those participated data owners during the query process is not effectively explored. (For those who did enable analytic obliviousness [65, 81, 85], they mainly rely on trusted hardwares and focus on improving security by addressing advanced side-channel problems therein.) The property of metadata-hiding in Vizard is inspired by emerging oblivious storage or file sharing systems [25, 34, 40, 52] that aim to hide the file access and sharing patterns. But different from their focus, Vizard only delegates usage rights of those outsourced data, and the consumers can only obtain an aggregated and metadata-hiding view of those contributed data.

Another line of works focuses on enabling owner-centric privacy policies (i.e., data policies in Vizard), so that owners can better enforce the usage of their data against a distrustful party. While most existing works enforce such policies via trusted parties or hardwares, Burkhalter *et al.* [19] explore to cryptographically enforce those privacy policies via a separate control layer that is comprised of decentralized committee nodes. They enforce that a given result can be correctly decrypted only if the underlying computation layer behaves faithfully and enforces all owners' privacy policies. However, they have not considered protection for the policies and thus could incur severe security threats due to metadata leakage (e.g., which owners are involved in a sensitive task.)

DPF-based Oblivious Systems. Since the introduction of functional secret sharing (FSS) and the effective distributed point function (DPF) constructions [14, 15], DPF has been widely studied to build oblivious messaging systems [28, 40], new private information retrieval (PIR) schemes [77], new scalable ORAM executions [34], and many others [12, 32, 61]. Vizard extends the scalable ORAM design proposed in [34] to craft optimized storage structures for data streams and explores other enriched policy control supports to improve Vizard's functionalities.

Release Policy Enforcements. Controlling when a generated result can be released has its necessity in various scenarios where owners want an extra layer of protection besides data confidentiality (e.g., execution correctness and result privacy), as firstly introduced by Zheng *et al.* [86] for machine learning tasks. But while today's works on verifiable computation [7, 8, 63, 68] and differential privacy [12, 24, 35, 84] can provide such a layer of protection, they usually rely on centralized trust (via one or few trusted auditors and noise generators) for determining whether the release policies are fulfilled or not. In contrast, Vizard designs useful release policies for private analytics and enforces them with decentralized trust using a byzantine-secure committee.

10 CONCLUSION

Vizard is a metadata-hiding data analytic system that allows data owners to share their data for collective knowledge in a streamlined, privacy-preserving, and fully-controlled manner. We hope this new system can help break the troublesome data silos problem today and facilitate large-scale data stream analytic services. Vizard's code is available on https://github.com/arimitx/vizard.

ACKNOWLEDGMENTS

We sincerely thank all anonymous reviewers for their useful comments and instructions. This work was funded in part by the Research Grants Council of Hong Kong under Grants CityU 11217819, 11217620, 11218521, 11202419, N_CityU139/21, RFS2122-1S04, C2004-21GF, R1012-21, and R6021-20F, and by the National Natural Science Foundation of China under Grants U20B2049 and U21B2018, by InnoHK initiative, the Government of the HKSAR, and Laboratory for AI-Powered Financial Technologies.

REFERENCES

[1] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. 2018. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX.. In *Proc. of NDSS*.
[2] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. 2018. PIR with Compressed Queries and Amortized Query Processing. In *Proc. of IEEE S & P*.
[3] Yoshinori Aono, Takuya Hayashi, Lihua Wang, Shiho Moriai, et al. 2017. Privacy-preserving deep learning via additively homomorphic encryption. *IEEE TIFS* 13, 5 (2017), 1333–1345.
[4] Borja Balle, James Bell, Adrià Gascón, and Kobbi Nissim. 2020. Private Summation in the Multi-Message Shuffle Model. In *Proc. of ACM CCS*.
[5] Raef Bassily and Adam Smith. 2015. Local, private, efficient protocols for succinct histograms. In *Proc. of ACM STOC*.
[6] Donald Beaver. 1991. Efficient Multiparty Protocols Using Circuit Randomization. In *Proc. of CRYPTO*.
[7] Mihir Bellare, Shafi Goldwasser, Carsten Lund, and Alexander Russell. 1994. Efficient probabilistic checkable proofs and applications to approximation. In *Proc. of ACM STOC*.
[8] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In *Proc. of CRYPTO*.
[9] Java Micro benchmark Harness. 2021. (2021). Online at: https://openjdk.java.net /projects/code-tools/jmh/.
[10] Iddo Bentov, Yan Ji, Fan Zhang, Lorenz Breidenbach, Philip Daian, and Ari Juels. 2019. Tesseract: Real-Time Cryptocurrency Exchange Using Trusted Hardware. In *Proc. of ACM CCS*.
[11] Blockcypher. 2021. (2021). Online at: https://www.blockcypher.com/.
[12] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. 2021. Lightweight Techniques for Private Heavy Hitters. In *Proc. of IEEE S&P*.
[13] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. 2021. Function Secret Sharing for Mixed-Mode and Fixed-Point Secure Computation. In *Proc. of EUROCRYPT*.
[14] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2015. Function Secret Sharing. In *Proc. of EUROCRYPT*.
[15] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2016. Function Secret Sharing: Improvements and Extensions. In *Proc. of ACM CCS*.
[16] Elette Boyle, Niv Gilboa, Yuval Ishai, Huijia Lin, and Stefano Tessaro. 2018. Foundations of Homomorphic Secret Sharing. In *Proc. of ITCS*.
[17] Benedikt Bünz, Lucianna Kiffer, Loi Luu, and Mahdi Zamani. 2020. Flyclient: Super-Light Clients for Cryptocurrencies. In *Proc. of IEEE S&P*.
[18] Lukas Burkhalter, Anwar Hithnawi, Alexander Viand, Hossein Shafagh, and Sylvia Ratnasamy. 2020. TimeCrypt: Encrypted Data Stream Processing at Scale with Cryptographic Access Control. In *Proc. of USENIX NSDI*.
[19] Lukas Burkhalter, Nicolas Küchler, Alexander Viand, Hossein Shafagh, and Anwar Hithnawi. 2021. Zeph: Cryptographic Enforcement of End-to-End Data Privacy. In *Proc. of OSDI*.
[20] Chengjun Cai, Lei Xu, Anxin Zhou, Ruochen Wang, Cong Wang, and Qian Wang. 2020. EncELC: Hardening and Enriching Ethereum Light Clients with Trusted Enclaves. In *Proc. of IEEE INFOCOM*.
[21] Chengjun Cai, Yifeng Zheng, Anxin Zhou, and Cong Wang. 2019. Building a Secure Knowledge Marketplace over Crowdsensed Data Streams. *IEEE TDSC* 1, 1 (2019), 1–1.
[22] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2019. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *Proc. of IEEE Euro S & P*.
[23] Hao Chen, Ilaria Chillotti, Yihe Dong, Oxana Poburinnaya, Ilya P. Razenshteyn, and M. Sadegh Riazi. 2020. SANNS: Scaling Up Secure Approximate k-Nearest Neighbors Search. In *Proc. of USENIX Security*.
[24] Rui Chen, Qian Xiao, Yu Zhang, and Jianliang Xu. 2015. Differentially Private High-Dimensional Data Publication via Sampling-Based Inference. In *Proc. of ACM SIGKDD*.
[25] Weikeng Chen and Raluca Ada Popa. 2020. Metal: A Metadata-Hiding File-Sharing System. In *Proc. of NDSS*.
[26] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D. Garcia. 2021. VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface. In *Proc. of USENIX Security*.
[27] Henry Corrigan-Gibbs and Dan Boneh. 2017. Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. In *Proc. of USENIX NSDI*.
[28] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. 2015. Riposte: An Anonymous Messaging System Handling Millions of Users. In *Proc. of IEEE S&P*.
[29] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* (2016). https://eprint.iacr.org/2016/086
[30] Ivan Damgård, Daniel Escudero, Tore Kasper Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. 2019. New Primitives for Actively-Secure MPC over Rings with Applications to Private Machine Learning. In *Proc. of IEEE S&P*.
[31] Leo de Castro and Antigoni Polychroniadou. 2021. Lightweight, Verifiable Function Secret Sharing and its Applications. *IACR Cryptol. ePrint Arch.* (2021). https://eprint.iacr.org/2021/580
[32] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. 2018. PIR-PSI: Scaling Private Contact Discovery. *PoPETs* 2018, 4 (2018), 159–178.
[33] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. 2004. Tor: The Second-Generation Onion Router. In *Proc. of USENIX Security*.
[34] Jack Doerner and Abhi Shelat. 2017. Scaling ORAM for Secure Computation. In *Proc. of ACM CCS*.
[35] Huayi Duan, Yifeng Zheng, Yuefeng Du, Anxin Zhou, Cong Wang, and Man Ho Au. 2019. Aggregating crowd wisdom via blockchain: A private, correct, and robust realization. In *Proc. of IEEE PerCom*.
[36] J. C. Duchi, M. I. Jordan, and M. J. Wainwright. 2013. Local privacy and statistical minimax rates. In *Proc. of IEEE FOCS*.
[37] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. 2006. Our Data, Ourselves: Privacy Via Distributed Noise Generation. In *Proc. of EUROCRYPT*.
[38] Stefan Dziembowski, Grzegorz Fabiański, Sebastian Faust, and Siavash Riahi. 2021. Lower bounds for off-chain protocols: Exploring the limits of plasma. In *Proc. of ITCS*.
[39] Eslam Elnikety, Aastha Mehta, Anjo Vahldiek-Oberwagner, Deepak Garg, and Peter Druschel. 2016. Thoth: Comprehensive Policy Compliance in Data Retrieval Systems. In *Proc. of USENIX Security*.
[40] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. 2021. Express: Lowering the Cost of Metadata-hiding Communication with Cryptographic Privacy. In *Proc. of USENIX Security*.
[41] Amazon Managed Streaming for Apache Kafka. 2021. (2021). Online at: https: //aws.amazon.com/msk/.

[42] Spring for Apache Kafka Framework. 2021. (2021). Online at: https://spring.io/projects/spring-kafka.
[43] Spring Framework. 2021. (2021). Online at: https://spring.io/.
[44] Badih Ghazi, Pasin Manurangsi, Rasmus Pagh, and Ameya Velingker. 2020. Private Aggregation from Fewer Anonymous Messages. In *Proc. of EUROCRYPT*.
[45] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proc. of SOSP*.
[46] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)* 43, 3 (1996), 431–473.
[47] Philippe Golle and Ilya Mironov. 2001. Uncheatable Distributed Computations. In *Proc. of CT-RSA*.
[48] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. 2012. Secure two-party computation in sublinear (amortized) time. In *Proc. of ACM CCS*.
[49] Slawomir Goryczka and Li Xiong. 2017. A Comprehensive Comparison of Multi-party Secure Additions with Differential Privacy. *IEEE Trans. Dependable Secur. Comput.* 14, 5 (2017), 463–477.
[50] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath T. V. Setty, Lorenzo Alvisi, and Michael Walfish. 2016. Scalable and Private Media Consumption with Popcorn. In *Proc. of USENIX NSDI*.
[51] Michael Hill and Dan Swinhoe. 2021. The 15 biggest data breaches of the 21st century. (2021). Online at: https://www.csoonline.com/article/2130877/the-biggest-data-breaches-of-the-21st-century.html.
[52] Yuncong Hu, Sam Kumar, and Raluca Ada Popa. 2020. Ghostor: Toward a Secure Data-Sharing System from Decentralized Trust. In *Proc. of USENIX NSDI*.
[53] DPF implementation. 2019. (2019). Online at: https://github.com/dkales/dpf-cpp.
[54] Apache Kafka. 2021. (2021). Online at: https://kafka.apache.org/.
[55] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *Proc. of IEEE S&P*.
[56] Steve Lu and Rafail Ostrovsky. 2013. Distributed Oblivious RAM for Secure Two-Party Computation. In *Proc. of TCC*.
[57] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. 2015. Privacy and Access Control for Outsourced Personal Records. In *Proc. of IEEE S&P*.
[58] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. 2017. Maliciously Secure Multi-Client ORAM. In *Proc. of ACNS*.
[59] Sinisa Matetic, Karl Wüst, Moritz Schneider, Kari Kostiainen, Ghassan Karame, and Srdjan Capkun. 2019. BITE: Bitcoin Lightweight Client Privacy using Trusted Execution. In *Proc. of USENIX SECURITY*.
[60] Miti Mazmudar and Ian Goldberg. 2020. Mitigator: Privacy policy compliance using trusted hardware. *PoPETs* 2020, 3 (2020), 204–221.
[61] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. 2020. Delphi: A Cryptographic Inference Service for Neural Networks. In *Proc. of USENIX Security*.
[62] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A system for scalable privacy-preserving machine learning. In *Proc. of IEEE S&P*.
[63] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2013. Pinocchio: Nearly Practical Verifiable Computation. In *Proc. of IEEE S&P*.
[64] Pew Research Center. 2019. Americans and Privacy: Concerned, Confused and Feeling Lack of Control Over Their Personal Information. (2019). Online at: https://www.pewresearch.org/internet/2019/11/15/americans-and-privacy-concerned-confused-and-feeling-lack-of-control-over-their-personal-information/.
[65] Rishabh Poddar, Ganesh Ananthanarayanan, Srinath Setty, Stavros Volos, and Raluca Ada Popa. 2020. Visor: Privacy-Preserving Video Analytics as a Cloud Service. In *Proc. of USENIX Security*.
[66] Edo Roth, Daniel Noble, Brett Hemenway Falk, and Andreas Haeberlen. 2019. Honeycrisp: large-scale differentially private aggregation without a trusted core. In *Proc. of ACM SOSP*.
[67] Edo Roth, Hengchu Zhang, Andreas Haeberlen, and Benjamin C. Pierce. 2020. Orchard: Differentially Private Analytics at Scale. In *Proc. of USENIX OSDI*.
[68] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy data analytics in the cloud using SGX. In *Proc. of IEEE S&P*.
[69] Shayak Sen, Saikat Guha, Anupam Datta, Sriram K. Rajamani, Janice Y. Tsai, and Jeannette M. Wing. 2014. Bootstrapping Privacy Compliance in Big Data Systems. In *Proc. of IEEE S&P*.
[70] Adi Shamir. 1979. How to share a secret. *CACM* 22, 11 (1979), 612–613.
[71] Elaine Shi, TH Hubert Chan, Eleanor Rieffel, Richard Chow, and Dawn Song. 2011. Privacy-preserving aggregation of time-series data. In *Proc. of NDSS*.
[72] Radu Sion. 2005. Query Execution Assurance for Outsourced Databases. In *Proc. of ACM VLDB*.
[73] Markus Stadler. 1996. Publicly Verifiable Secret Sharing. In *Proc. of EUROCRYPT*.
[74] Florian Tramèr, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. 2017. Sealed-Glass Proofs: Using Transparent Enclaves to Prove and Sell Knowledge. In *Proc. of IEEE EuroS&P*.

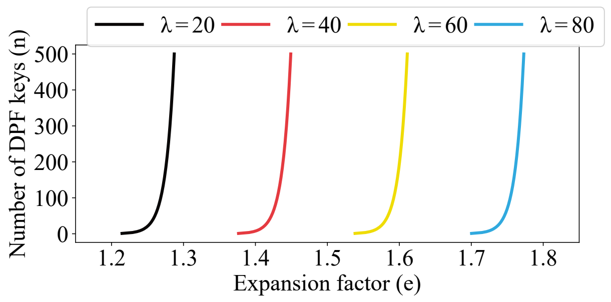


Figure 12: Given $p = 3$ salted random mapping functions, this figure shows the requirement on the expansion factor e to achieve security with varying parameter settings, i.e., $\lambda = 40, 60, 80$, and 100.

[75] U.S. Department of Health and Human Services Office for Civil Rights. 2021. Breach Portal: Notice to the Secretary of HHS Breach of Unsecured Protected Health Information. (2021). Online at: https://ocrportal.hhs.gov/ocr/breach/breach_report.jsf.
[76] Frank Wang, Ronny Ko, and James Mickens. 2019. Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services. In *Proc. of USENIX NSDI*.
[77] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. 2017. Splinter: Practical Private Queries on Public Data. In *Proc. of USENIX NSDI*.
[78] Ning Wang, Xiaokui Xiao, Yin Yang, Jun Zhao, Siu Cheung Hui, Hyejin Shin, Junbum Shin, and Ge Yu. 2019. Collecting and analyzing multidimensional data with local differential privacy. In *Proc. of IEEE ICDE*.
[79] T. Wang, J. Blocki, N. Li, and S. Jha. 2017. Locally differentially private protocols for frequency estimation. In *Proc. of USENIX Security*.
[80] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* 151 (2014), 1–32.
[81] Min Xu, Antonis Papadimitriou, Ariel Feldman, and Andreas Haeberlen. 2018. Using Differential Privacy to Efficiently Mitigate Side Channels in Distributed Analytics. In *Proc. of the 11th European Workshop on Systems Security, (EuroSec)*.
[82] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. 2016. Revisiting Square-Root ORAM: Efficient Random Access in Multi-party Computation. In *Proc. of IEEE S&P*.
[83] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. 2018. RapidChain: Scaling Blockchain via Full Sharding. In *Proc. of ACM CCS*.
[84] Xiaojian Zhang, Rui Chen, Jianliang Xu, Xiaofeng Meng, and Yingtao Xie. 2014. Towards Accurate Histogram Publication under Differential Privacy. In *Proc. of SIAM International Conference on Data Mining*.
[85] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *Proc. of USENIX Symposium on Networked Systems Design and Implementation, NSDI*.
[86] Wenting Zheng, Ryan Deng, Weikeng Chen, Raluca Ada Popa, Aurojit Panda, and Ion Stoica. 2021. Cerebro: A Platform for Multi-Party Cryptographic Collaborative Learning. In *Proc. of USENIX Security*.
[87] Shoshana Zuboff. 2019. The Age of Surveillance Capitalism. *Profile Books* (2019).

A ALTERNATIVE SOLUTIONS

Below we discuss some potentially feasible alternative solutions that can fulfill Vizard's security goals.

Generic MPC with ORAM. One apparent solution to fulfill our goal of a policy-controlled metadata-hiding data analytic system is to leverage existing generic-purpose secure two-party computation (S2PC) for RAM programs (e.g., [23, 25, 34, 48, 56, 82]). Each owner sends a share of his data and policy to each server, and then the two servers can jointly run an S2PC process of a RAM program to 1) combine the shares; 2) securely find out the matching data based on the policy and query request; and 3) aggregate the matched data and output the result. However, in order to preserve metadata (i.e., access pattern) protection against the two servers, we have to additionally construct an ORAM client inside the S2PC, which would incur substantial costs and affect efficiency when handling a large amount of data [34]. In contrast, although our approach falls in the same two-server setting, we design non-trivial optimizations to boost the efficiency of the S2PC process and avoid the need of an S2PC-based ORAM client for hiding data access patterns.

Secure Aggregation with DP Privacy. Another line of works on privacy-preserving data analytics has extensively explored computing statistics in either the local model [5, 36, 78, 79] (to list just a few) or the shuffle/anonymous model [4, 44] of differential privacy (DP). Each owner sends his data (with DP protections) to the data consumers, who can then obtain the desired results (e.g., summations) by readily computing on those (noisy) data. But while they can construct protocols without the need of non-colluding servers, they would either leak a non-negligible amount of private information about the data owner's data streams or could largely affect the accuracy of the results [12]. In contrast, our solution provides much stronger privacy guarantees for the data owners, and we can also enforce DP privacy (as part of the release policy) for the result outputs.

Trusted Hardwares. The use of trusted hardwares (e.g., Intel SGX [29]) for secure data analytics [35, 68] has its unreachable advantages on efficiency. Each owner securely sends his data streams and policies to the enclave through an encrypted channel, and the enclave can process them as plaintext for generating the results (along with integrity guarantees from the trusted hardware itself). Due to self-limited memory sizes, the uploaded data streams might be stored outside the enclave, and thus we need to establish an additional ORAM client inside the enclave [1] for securely accessing the outside storage. Besides, recent attacks have also shown potential confidentiality leakages through measuring enclave execution patterns [22] or other hardware-based measurements [26], which could largely undermine enclave security. In contrast, our solution does not rely on trusted hardwares.

B ANALYSIS FOR OR OPERATORS

Recall that in our construction for supporting OR operators, we propose to leverage a random mapping function $\mathsf{RM} : \{0,1\}^{\lambda} \times \{0,1\}^{N} \rightarrow [m]$ with p different salts to securely place n DPF keys, so that later the server will need only to evaluate p DPF keys to compute the controlling value. With this setting, it is readily understood that we want to reduce the parameter p, so that each server will do fewer computations (which is also the goal we want to achieve in the first place). However, given that we only have p ($p \leq n$) salted random mapping functions to place those n keys, a smaller number of p would inevitably increase the failure probability (i.e., we cannot successfully find positions for all those n keys). In essence, this failure probability problem has been extensively studied in the cuckoo hashing paradigm. Below we will sketch some of the recent results from Demmler *et al.* [32] to justify our choice of the parameters p and m. Let e be an expansion factor such that $m = e \cdot n$, and λ be a security parameter such that we will have $2^{-\lambda}$ probability of failure in the above key inserting task. The empirical analysis from [32] demonstrates that

$$\lambda = a_n \cdot e + b_n,$$

where $a_n \approx 123.5$ and $b_n \approx -130 - \log_2 n$, for $n \leq 512$ and $p = 3$. From Figure 12, we can see that, in order to maintain constant computation cost on the server sides, the expansion factor e would grow with the increase of the security parameter λ that we want to achieve. This indicates that a data owner might need to create more fake DPF keys to fill in the unfilled positions in those m buckets, which would incur an increase of the communication cost for key transmission.

But we can observe that the increase of the underlying DPF keys to be placed (i.e., value n) does not impact too much on the expansion factor, which indicates that our solution would bring a constant scale of fake DPF keys overhead even though we are dealing with an increased number of DPF keys. Here in our system, we will allow a maximum of $n = 50$ OR operations per data policy (which we think should be enough) and adopt $\lambda = 40$ as our security parameter [2] for $p = 3$ salted random mapping functions. Accordingly, the total buckets sent to the two servers for OR operators will be $m = 1.5 \cdot n$, where n is the number of DPF keys an owner wants to insert.

C RELEASE POLICY CONSTRUCTIONS

In this section, we will demonstrate the design details of the result release policies that can be supported in Vizard.

C.1 Integrity-based Release Policies

Here, we will only focus on a "ringer"-based integrity checking design in this section and omit the concrete designs via publicly verifiable MPCs (as existing designs can be readily adopted in our two-server setting for checking output results [12]). Now, we first summarize how the "ringer" technique works using a basic scheme proposed in [72] that aims to bring execution assurance for an outsourced database.

Basic Ringer Constructions. Consider a set of queries $\{q_1, \ldots, q_k\}$ that should be executed via a function $F(\cdot)$, we first randomly select a secret query (e.g., $q_m, m \in [k]$) and compute a challenge token (named "ringer") from the selected query as $H(F(q_m))$, where H is a one-way function like cryptographic hash. Then, the challenge token and the set of queries will be sent to a service provider. To prove it has correctly performed the required function $F(\cdot)$ for all queries, in addition to the query results, it also needs to return a proof m^* for integrity checking. Thanks to the one-way property of $H(\cdot)$, if $m^* = m$, we can assure that the service provider has correctly processed this set of queries.

Our Treatments. Observe that there is no easy way for the RRC to calculate a challenge token, as owners' data policies and data values are not shared with the RRC, we thus turn our focus to explore owner-assist approaches. Here, our idea is to let owners periodically and randomly select a representative that will insert a specially-crafted query q^*, whose challenge token is pre-computed by all owners, to Vizard for integrity checking. Specifically, for each selected representative, our treatment works in two consecutive stages, i.e., *pre-computation* and *challenge*.

At a high level, in the pre-computation stage, the representative will generate a large number of challenge queries to the owners and obtain their associated challenge tokens. Then, the representative will periodically inject a randomly selected challenge query to a batch of queries that will be executed by the two servers and secretly share his true answer to the RRC. Finally, each trustee in the RRC can verify the integrity of the results delivered by the two servers by 1) recovering the true answer given by the representative, and 2) checking whether the two servers have correctly found the "ringer" query or not. Below are some security concerns we need to further address on top of this treatment workflow.

Security Considerations. Firstly, for a large set of queries $\{q_1^*, q_2^*, \dots\}$ generated by a randomly selected owner representative, each owner i will be able to match his data policy P_i to each query and determine whether they should participate or not. But apparently, the absence of data value of a given query will leak information about his data policy to the representative, so we ask each owner to submit homomorphic commitments of his data value and submit a commitment of zero if the data is not involved in a given query. Since the commitments submitted by all owners are additive, the representative can readily generate the challenge token of a query q^* in the form of $\mathsf{Com}(F(q^*))$ (where Com is a homomorphic commitment scheme like Pedersen commitment) by combining all owners' commitments. Besides, it allows the two servers to securely evaluate whether a query result matches $\mathsf{Com}(F(q^*))$ (so as to find the "ringer" query) by locally computing a commitment of its result output and sending it to the other server.

Secondly, it is well understood that the integrity checking treatment above relies on the secrecy of those challenge queries (i.e., the two servers do not know which one is the challenge query in advance). Therefore, the answer to a challenge token will be sent to the RRC in secret sharing forms, so that the two servers cannot learn about the answer without correctly executing those queries (e.g., by compromising a small set of trustees inside the RRC). Specifically, we will share those secrets using Shamir's secret sharing scheme [70] and set the recovery threshold as $t = 2/3$ (which meets the security threshold we illustrated in Section 2.2).

Lastly, to avoid being detected as a representative that will perform integrity checking, the selected owner might need to create consumer accounts for submitting his challenge queries, or it can contact some existing consumers to help them insert those challenge queries. To ease their operation costs, Vizard can provide a trusted shuffling service (e.g., through a mixed net enabled by the RRC) to break the links between queries and their submitters. Here to keep our protocol concise and clear, we adopt the basic construction (i.e., let the representative create consumer accounts) in our implementations for demonstration purposes.

Fake Challenge Tokens. One obvious problem of our approach is that it can only achieve probabilistically-secure integrity checking, and the two servers can still stop processing as soon as they have found the "ringer" query inside. One strategy to reduce the false positive (FP) rate defined above is to ask the representative to send multiple challenge queries and tokens to the two servers. Besides, we can also let the representative generate fake challenge tokens and send them to the two servers. Fake challenge tokens do not match any result in the batch of queries to be executed, and they are just random values generated by the representative. Therefore, the cost for fake challenge token generation is very cheap.

The existence of more challenge tokens (thanks to those cheap fake challenge tokens) makes the two servers more hesitant to stop even they have found some "ringers", as they don't know whether there are other true "ringers" in the remaining challenge tokens. As illustrated in [72], given a batch of 20 queries that consist of 2 ringers and 4 fake challenge tokens, the server will have a low success rate (less than 20%) even after executing 16 queries.

C.2 Privacy-based Release Policies

Recall that in Vizard, we also allow privacy-based release policies that aim to correctly generate differential privacy noise and add it to a query result, i.e., $\mathsf{out}_0 + \mathsf{out}_1 + \mathbf{noise}$, before deliveries. Specifically, the generated noise can be added to the result during the query process by the two servers, or by trustees in the RRC after receiving result shares given by the two servers. But since Shamir's secret sharing scheme does not support homomorphic operations on the shares, in Vizard we adopt the former approach and add noise to the result during the query process. The generated (DP-noised) result share on each server will also be split and shared to trustees in the RRC for deliveries, so that we can enforce other policies (e.g., integrity and payment) at the same time.

We note that some recent works have proposed homomorphic secret sharing schemes (HSS) [16] that allow share operations, but their current constructions would require expensive overhead for both share generations and recoveries, making them impractical to be used for handling large-scale queries. We leave a concrete and optimized design that is based on HSS (on the RRC side) in our future works.

Concrete Design. Our goal here is to enforce that a correct amount of DP noises (i.e., fits the (ϵ, δ) privacy budget that is jointly defined by owners) will be added during the query process with the help of a decentralized committee. Fortunately, we can follow the line of byzantine-secure distributed noise generation works [37, 71] and let each trustee locally generate noise. Specifically, each trustee generates a noise drown from a carefully-parameterized geometric distribution (i.e., $\mathsf{Geom}(\exp(\frac{\epsilon}{SR}))$, with probability $\frac{1}{t \cdot W} \log \frac{1}{\delta}$), where SR is the share output range of the underlying secret sharing scheme and t ($t = 2/3$ in Vizard) is the byzantine threshold. Each generated noise will be securely split and given to the two servers, where every trustees' noise shares will be accumulated on each server and added to its result share for protection. Each server will then securely split its result share to each trustee in the RRC (along with integrity proofs) as we demonstrated before for the release process. This method can achieve (ϵ, δ)-DP privacy for the query result (with a small error of roughly $O(\frac{SR}{\epsilon}\sqrt{\frac{1}{t}})$ if $t = 2/3$ trustees in RRC are honest). We note recent works [4, 49] have further reduced the total noise needed, and their constructions are also applicable to Vizard.

Remarks on Privacy Budget. When the budget is exhausted, no more query should be conducted for avoiding unintended privacy leakage. This indicates that we might need to replenish the budget regularly (e.g., daily) to keep our query service operational. But we also note that a recent work [66] has explored to use the Sparse-Vector technique to stretch a given privacy budget. It would be interesting to explore whether this technique can be used in Vizard to prolong the life-circle of our query service, and we leave such exploration in our future works.

C.3 Payment-based Release Policies

Vizard allows data owners to specify payment-based release policies that enforce monetary rewards for them before the result is delivered to the consumers. Here, given the fact that the result is secret shared and requires the majority of trustees (e.g., 2/3) to recover, an apparent idea is thus to ask each trustee in the RRC to serve as an arbiter to enforce payment-based result delivery, so that a malicious consumer cannot obtain enough result shares to recover the result if he has not faithfully made a payment. Here to ensure that the payment record can be publicly verifiable by each trustee, Vizard turns to the emerging blockchain technologies and crafts a smart contract-based payment channel on Ethereum [80] for consumers to make payments and show proofs to trustees in the RRC. The process executes as follows:

(a) Vizard will initialize a smart contract SC, and ask data owners to create on-chain accounts and record their accounts to SC for receiving the payments.
(b) Data owners can then jointly decide their payment policies (e.g., deciding $Price for each specific type of query task).
(c) After making a query to our system, each consumer will obtain a query id qid that will be used to associate his on-chain payments.
(d) To make a payment, the consumer will send a payment transaction that records ("qid", "$Price") to pay a pre-defined amount of money to SC, and this payment will be evenly split to all registered data owners. (To support more fine-grained payment strategies, e.g., quality-based, without leaking the reward distributions, we can further adopt off-chain payment channels [10, 38] to deliver owners' rewards.)
(e) The consumer can now contact each trustee in the RRC with a payment proof for result retrievals, and each trustee will check the proof and (if the proof is validated) deliver its local result share to the consumer.

Remarks. The use of blockchain to construct an on-chain payment channel can ensure public verifiability, but it would also incur enormous blockchain maintenance overhead for each trustee (e.g., require storing over hundreds of GBs of data to become a full node).

To relieve such maintenance overhead, we can further leverage existing light client designs, which store only lightweight block headers and rely on an existing full node for transaction retrievals. Here in Vizard, we will ask each trustee to use existing blockchain infrastructure (e.g., Infura for Ethereum) for simplicity, and we are aware that additional integrity hardening techniques (e.g., [17, 20, 59]) can be further adopted to enforce security.