Data Station: Delegated, Trustworthy, and Auditable Computation to Enable Data-Sharing Consortia with a Data Escrow

Siyuan Xia, Zhiru Zhu, Chris Zhu, Jinjin Zhao, Kyle Chard, Aaron J. Elmore, Ian Foster, Michael Franklin, Sanjay Krishnan, Raul Castro Fernandez
The University of Chicago
{stevenxia,zhiru,chz,j2zhao,chard,aelmore,foster,mjfranklin,skr,raulcf}@uchicago.edu

ABSTRACT

Pooling and sharing data increases and distributes its value. But since data cannot be revoked once shared, scenarios that require controlled release of data for regulatory, privacy, and legal reasons default to not sharing. Because selectively controlling what data to release is difficult, the few data-sharing consortia that exist are often built around data-sharing agreements resulting from long and tedious one-off negotiations.

We introduce Data Station, a data escrow designed to enable the formation of data-sharing consortia. Data *owners* share data with the escrow knowing it will not be released without their consent. Data *users* delegate their computation to the escrow. The data escrow relies on delegated computation to execute queries without releasing the data first. Data Station leverages hardware enclaves to generate *trust* among participants, and exploits the centralization of data and computation to generate an audit log.

We evaluate Data Station on machine learning and data-sharing applications while running on an untrusted intermediary. In addition to important qualitative advantages, we show that Data Station: i) outperforms federated learning baselines in accuracy and runtime for the machine learning application; ii) is orders of magnitude faster than alternative secure data-sharing frameworks; and iii) introduces small overhead on the critical path.

PVLDB Reference Format:
Siyuan Xia, Zhiru Zhu, Chris Zhu, Jinjin Zhao, Kyle Chard, Aaron J. Elmore, Ian Foster, Michael Franklin, Sanjay Krishnan, Raul Castro Fernandez. Data Station: Delegated, Trustworthy, and Auditable Computation to Enable Data-Sharing Consortia with a Data Escrow. PVLDB, 15(11): 3172 - 3185, 2022.
doi:10.14778/3551793.3551861

1 INTRODUCTION

Pooling and sharing data increases and distributes its value. Organizations that pool their data can build and mutually benefit from more powerful machine learning models [34]. Health organizations that share data with each other can improve patient care [42]. And, researchers who share experimental data can accelerate scientific discovery [55]. Despite the obvious advantages, few data-sharing consortia form in practice. Many organizations that could benefit from data sharing face regulatory, legal, privacy, incentive, and technical barriers, and thus, can only release selected data in a controlled manner [4, 13]. Technically speaking, controlling how data is used is difficult, so many beneficial data-sharing consortia never materialize. Those that do are often built around data-sharing agreements resulting from long and tedious one-off negotiations that are inflexible to later changes in how data should be used.

In this paper, we introduce Data Station, an intermediary **data escrow**, the computational and data management infrastructure designed to enable the formation of data-sharing consortia. Data *owners* share data with the escrow as if it was an extension of their own infrastructure, i.e., it can be guaranteed that their data will remain confidential and that no one will access it (or any derived insights) without their explicit permission. Data *users* who want to extract insights from data *delegate* their computation to the escrow, and that computation will be executed only if permitted by the data owners. The escrow ensures that all data is protected, makes few assumptions on the threat model, and thus, allows owners and users to *trust* it. Finally, because many sharing scenarios involve regulatory and compliance requirements, all computation that takes place on the platform must be transparent so third-party auditors and compliance officers can *audit* the consortia.

1.1 Data Sharing Scenarios

We offer stylized scenarios based on examples of real applications to illustrate the opportunities of enabling data-sharing consortia.

1.1.1 Data Sharing Within Organizations: Team Consortia. When analysts set out to solve a data task, such as building a machine learning model, or extracting the results of a query to complete a report, they have to find relevant data among a myriad of data sources within their organization [17]. Unfortunately, many of these data sources are siloed and are managed by individual data owners whose responsibility is to control who accesses the data. This introduces an important challenge. Analysts do not necessarily know whether a dataset is useful for their task before seeing it, so they must work with owners in time-consuming one-off negotiations to understand the dataset and negotiate access. Furthermore, even after securing access to a dataset, analysts may find that it is incomplete or poorly described, or that access to additional datasets is necessary, producing a back and forth cycle that increases the time to insights. In short, even within an organization, analysts must invest a significant upfront cost to determine if data is useful. Owners must resort to conversations with the analysts to understand whether data access should be granted.

What would help in this scenario is a platform that permits evaluation of the analysts' tasks on the owners' data without owners

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551861

having to release the data first. A data escrow would enable owners and users to address the problem while also allowing compliance officers to audit how employees are using data assets and rapidly detect misuse, e.g., training ML models over sensitive attributes.

1.1.2 Data Sharing Across Organizations: Organization Consortia. Many organizations would mutually benefit from pooling their data to train better machine learning models, but they are wary of sharing their raw datasets. For example, chemical engineering organizations may be willing to pool expensive-to-obtain simulation data to train more powerful models and find better materials [12], but they do not want other organizations to see the data they possess. They would be comfortable sharing inferences over a model trained on everybody's data, but there is no easy solution to orchestrate such data-sharing consortia that does not leak information about their individual datasets. Despite the benefits of sharing, the consequence of the above risks is that organizations do not share data and the value remains untapped.

What is needed is a platform that combines each participant's data without releasing it to anyone, trains a model, and selectively allows participants to access model inferences. Such a platform will help organizations pool their data and unleash its value.

1.2 The Data Station System

The main contribution is Data Station, a new data escrow system that enables delegated, trustworthy, and auditable computation.

Delegated Computation. Today, data access and processing are intertwined. To run computation, one has to access the data first. However, as data sharing is constrained by the barriers described in the examples, no computation takes place and no value is extracted. Data Station acts as an escrow to whom data owners send their data and data users send their computation. Delegating computation to Data Station means the Station can promise to protect data, i.e., no data or derived data products will be released without the data owners' explicit consent. Further, this model stops users from paying upfront costs to access data and allows them to concentrate on gaining access to their query results instead.

Trustworthy Computation. The introduction of an escrow enables data sharing as long as both owners and users trust it to keep their data secured. Users must trust that the escrow runs their computation securely and does not leak it to other participants. Owners must trust the escrow to honor their access preferences. Eliciting trust requires different mechanisms that depend on the threat model. Data Station implements a *full-trust* mode: useful in situations such as when employees perform data discovery within their own organization, that runs the intermediary; and *near-zero-trust* mode: useful in cases such as when independent organizations want to pool their data using a third-party intermediary. To implement these mechanisms, Data Station leverages secure hardware enclave technology [29] and cryptographic techniques. Unlike confidential computing approaches [5, 6, 9, 35] geared towards letting a user run computation on their own data but on a third-party infrastructure, e.g., a cloud vendor, Data Station is designed to run computation on data from multiple parties.

Auditable Computation. Many challenging data-sharing scenarios are regulated and subject to compliance and audit rules [15]. In these cases, even if owners and users trust each other, the computation they perform on data must be transparent to third parties, such as compliance officers and auditors. Data Station exploits the centralization of data and compute in the platform to record all computation in a tamper-proof immutable log. Every attempt to access data, every running task, as well as the data access preferences of data owners are stored in the log. This log lets authorized auditors verify that the tasks Data Station runs follow the compliance rules and regulations that govern the data.

Contribution and Evaluation Results. To the best of our knowledge, Data Station is the first data escrow system designed to enable data-sharing. In the evaluation, we show two sharing applications enabled by Data Station. First, we show that compared to a federated learning deployment, Data Station achieves much higher accuracy in a small portion of the time for several learning tasks. Second, we show how Data Station has up to two orders of magnitude lower overhead than alternative technologies that support end-to-end encryption, thus enabling a wider range of applications. Finally, we emphasize the qualitative advantages of Data Station and conduct a thorough evaluation of its overheads.

Focus of this Paper. Successfully forming data-sharing consortia requires consideration of aspects such as privacy constraints, regulations, legal data-sharing agreements, incentives among participants, and more. All these issues are important, but they matter only if there is a technical solution to share data in the first place. Data Station is designed to tackle the technical challenge.

The rest of the paper is organized as follows. Section 2 gives an overview of Data Station. Section 3 explains how Data Station achieves the goals of delegated and auditable computation. Section 4 explains how to support the *near-zero-trust* mode. Section 5 introduces the design and implementation of an execution environment for Data Station. Section 6 presents evaluation results, Section 7 the related work, and Section 8 the conclusions.

2 DATA STATION OVERVIEW

We present the abstractions used by Data Station in Section 2.1, the sharing lifecycle in Section 2.2 and the computation lifecycle in Section 2.3. Then, we state the promise Data Station makes in Section 2.4, and overview its architecture in Section 2.5.

2.1 Agents, Data Elements, Functions

An agent $a_i \in \mathcal{A}$ is any entity that interacts with Data Station. There are three types of agents. *Owners* control access to data assets they own. It may be beneficial to share these data assets with other agents, so owners will be willing to *register* them with Data Station. Registering a data asset with Data Station copies the data from the owners' infrastructure to Data Station's infrastructure. *Users* are agents who want to run computation on data that is registered with Data Station. Finally, *operators* are neither owners or users, but they want to understand what computation is running on what data inside Data Station. Auditors, compliance officers, and other kinds of regulators may play the role of operators.

Any registered data asset is represented in Data Station as a *data element* (DE), d_i. DEs include data of different types and granularities, such as relations, databases, files, images, and more.

All computation in Data Station is represented via *functions*, $f \in \mathcal{F}$. Data Station provides some basic functions, but most functions are provided by owners and users planning to form a sharing consortia, e.g., a coalition of chemical engineers may provide specialized indexes and search functionality for molecular data. These functions are exposed in Data Station via $f \in \mathcal{F}$. Functions take input parameters and DEs as input and produce other DEs and optionally other side effects, such as logs and temporal files, e.g., a train function takes input DEs as training data and produces a model. No function side effect is visible to users. It follows that functions should run end-to-end. Providing a suite of functions to prepare and integrate data using the escrow is possible, but that requires solving additional challenges out of the scope of this paper.

2.2 Policies and Sharing Modes

Owners register DEs with the Station with the intention of eventually letting some computation run on that data. Owners fully control for what purposes Data Station accesses the data they register via *policies* and *sharing modes*.

Policies. A policy is a triple, a_i, f_i, d_i that indicates that agent a_i can run function f_i on DE d_i. In contrast to access control policies that broker low-level operations to files, such as read, write, and execution permissions in the context of MAC in Unix-based systems, a policy in Data Station indicates what functions can run on what DEs, so it is a higher-level description.

Sharing Modes. There are three *sharing modes* that indicate what types of data access are available: *sealed*, *enclave*, *open*. In the *sealed* mode, a registered DE cannot be used by any function or by Data Station unless there is an explicit policy permitting such access. When there are no policies, owners who register a DE in *sealed* mode can think of Data Station as a mere extension of their own infrastructure, because no computation can take place and the existence of the DE is not disclosed to anyone.

In *enclave* mode, Data Station can run functions on the DE, but no output will be released without explicit consent from the owner, i.e., without an explicit policy permitting the release of the results. This mode permits Data Station to perform tasks such as index creation, profiling, training models, and more, while guaranteeing that no information is released to anyone.

Finally, we say a DE, d_j is in *open* mode for a given agent a_i when there is a policy that includes a_i and d_j.

Lifecycle. A data owner may initially register a DE in *sealed* mode, but owners register DEs with the intention of eventually allowing users to benefit from their existence. Owners may keep DEs in *sealed* mode and write policies to describe with fine-granularity who can run what computation on their data. This is useful, e.g., when allowing a third-party to test a piece of software on their data, or for secure data exchange via an intermediary. In other scenarios, owners may set the DEs to *enclave* mode, letting Data Station run computation on them while keeping results private. This is useful to, e.g., build indexes that permit users to discover relevant DEs but ensuring the actual data is never released without explicit consent from the owners. Ultimately, for a DE or derived data product to leave the Station, the data owner must have written a policy explicitly, so that the DE is in open mode.

2.3 Computation Lifecycle

Users invoke functions pre-registered in the Station, $f_i \in \mathcal{F}$. A function invocation triggers the creation of *intents*, which are triples defined analogously to policies a_i, f_i, d_i and that indicate the intention of agent a_i to execute function f_i on DE d_i. Intents are never created over *sealed* DEs unless there is a policy that permits the execution of that function. *Sealed* DEs are invisible to functions.

We differentiate between two broad classes of functions, *data-blind* and *data-aware* functions. The first kind does not require knowledge of any DE in Data Station. For example, search and query-by-example functions take input from a user who does not need to know about any DE. In contrast, *data-aware* functions take as a parameter a set of DEs. For example, a copy/download function needs to indicate what DE to download. It follows that no user can call data-aware functions on (effectively invisible) *sealed* DEs.

Derived Data Products. When a function runs on a DE and produces an output, we call this output a *derived data product*. A derived data product is another DE that resides inside Data Station. Hence, DEs can be uploaded by their owners, or produced by functions. Derived data products are a key ingredient of delegated computation, as they are the results for which users come to Data Station. One key challenge Data Station must solve is to apply policies created on DEs registered by owners to derived data products, even when these may have been derived from DEs owned by different owners.

2.4 Data Station's Promise and Trust Modes

Data Station promises owners that only DEs, including derived data products, that are in *open* mode ever leave Data Station. Furthermore, it promises owners that any activity involving DEs they own is recorded and visible to them on-demand.

Maintaining the guarantee requires different protocols, algorithms, and even infrastructure depending on the threat model. For example, when Data Station runs inside an organization to enable their employees to discover data assets owned by other teams, a reasonable threat model may be that the infrastructure where Data Station is deployed is non-adversarial, that DEs will be kept in their original *sharing mode*, and that the implementation follows the promise as specified. In contrast, when Data Station runs on third-party infrastructure, it must ensure the promise is kept in a more challenging threat model. Data Station operates on different modes to guarantee the promise under different threat models. Agents consider Data Station *trustworthy* when it keeps the promise under their target threat model.

2.5 Architecture Overview

An overview of Data Station architecture is shown in Fig. 1. Existing applications can be registered with Data Station via the App Register component that tells the Gatekeeper what functions are available for execution. Once registered, data users can invoke those functions using various interfaces that use the Agent APIs component. All function invocations are brokered by the Gatekeeper, which checks with the Policy Broker if the invocation can proceed, according to the policies present for the data involved and the respective sharing mode. Such policies are written by data owners using the Agent APIs. If the execution can proceed, the Gatekeeper

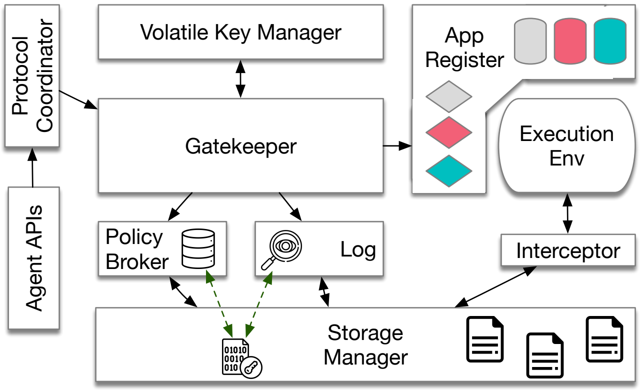

Figure 1: High-level overview of Data Station architecture

instantiates a task to execute that application, all using the execution environment, where the application runs in isolation. All accesses to the Storage Manager—whether to access existing data or to store derived data products or any other temporary data—is mediated by an Interceptor component. Finally, to support operation under different threat models, Data Station uses a Volatile Key Manager and a Protocol Coordinator that we describe later.

3 DELEGATED, AUDITABLE COMPUTATION

We introduce the main components of Data Station that permit delegated and auditable computation. In this section, we assume both owners and users trust the infrastructure, the function implementation, and the administrators who maintain the platform. We relax these assumptions in Section 4.

Goal. The primary goal of Data Station is to run functions invoked by users on data supplied by owners in a way that satisfies owners' sharing preferences, i.e., keeping Data Station's promise.

3.1 Gatekeeper In Depth

After a function invocation, Data Station identifies what DEs are accessible to the function, including derived data products. The Gatekeeper acts as a single-point-of-entry for all function invocations. Thus, it orchestrates all necessary steps to serve function calls while maintaining Data Station's promise. Users can only invoke functions exposed by the Gatekeeper. Functions are registered with the Gatekeeper through an application registration process (Section 5). Deciding what DEs are available to a function involves a different process for *data-aware* and *data-blind* functions.

Brokering Data-aware Functions. Data-aware functions request access to specific DEs via paths, names, or other identifiers. Ahead of executing the function, the Gatekeeper creates the corresponding intents by using the calling agent id, a_i, the function being invoked, f_j, and the DE being accessed, d_k. The Gatekeeper creates an intent per DE involved. Then, it uses the Policy Broker to determine whether the intents have matching policies. The Policy Broker component is backed up by a database that contains all agents, functions, DEs, and policies ever registered in the system. Then, given an intent, (a_i, f_j, d_k), the Policy Broker creates a query that checks whether there is a policy that permits a_i to execute f_j on d_k. For efficiency, whenever more than one intent is created, these are all represented in a single query. Finally, if there are matching policies, the Gatekeeper permits the function invocation on that DE. Otherwise, it blocks execution.

Brokering Data-blind Functions. *Data-blind* functions do not include a list of DEs the function needs to access, unlike *data-aware* functions. The Gatekeeper determines what DEs are accessible to the function and agent by combining: i) the set of DEs with a matching policy; ii) the set of DEs in *enclave* mode. Both sets are retrieved from the Policy Broker. The first is by querying DEs that match a predicate containing the calling agent, a_i, and function, f_j. The second is by requesting DEs stored in *enclave* mode.

Function Execution. After determining what DEs are visible to a function, the Gatekeeper must enforce the function only accesses those DEs. This is achieved by creating a "jailed" (i.e., in the Unix chroot [28] sense) execution environment that isolates the function from all available resources except for those it is given explicit permission. We explain the design of the execution environment in Section 5. For now, it suffices to know that such execution environment also captures: i) the concrete set of DEs the function *actually* accesses; and ii) the DEs it produces. First, note that a function may be given access to more DEs than it actually needs access to; for example, a function that builds a spatio-temporal index only needs access to DEs with spatial and temporal attributes. Second, a function will produce and return results. The Gatekeeper must manage the results returned by a function invocation.

Staging Zone and Result Delivery. The Gatekeeper must decide whether it can forward the results of a function invocation to the calling agent or not. If the function invoked is *data-aware*, then the function did execute because there is a matching policy for the specific DE indicated. If there is a matching policy, it means that the DE's owner indicated that a_i can execute f_j, which implies a_i can access the results. Thus, in this case the Gatekeeper will return the results to the calling agent. If the function invoked is *data-blind*, the function may have executed over both DEs for which there was a policy available, and *enclave* DEs for which there was no policy. When the results depend on only DEs for which there was a matching policy, the Gatekeeper can return them to the calling agent. When they also depend on *enclave* DEs, the results cannot be returned without previously obtaining permission from the *enclave* DE owner. The Gatekeeper determines whether the results depend on only DEs with matching policies or not by retrieving the list of actually accessed DEs from the execution environment, as explained above. Any results that cannot be delivered to a_i directly are stored in the *staging zone*, where they exist as *enclave* DEs, awaiting a matching policy that permits their release.

Result Granularity and Brokering Access. When an *enclave* DE, d_k, is part of a function's results, the Gatekeeper seeks permission from the data owner to release d_k to a_i. The owner originally set the DE in *enclave* mode with the intention of permitting Data Station to perform some computation, but without establishing what functions could be invoked and by whom. After the DE is part of the results of a function invocation, the Gatekeeper provides precise information on both who accessed the DE, a_i and via what function, f_j. The owner then may decide to grant access to the calling agent, which amounts to creating the policy (a_i, f_j, d_k).

Denying access leaves the list of policies unmodified and the DE removed from the staging zone. Finally, if the list of *enclave* DEs is large, brokering access, which requires involving data owners may become time-consuming. Data Station gives the calling agent the option to call the functions over only the collection of DEs for which a policy is available. This option allows calling agents who want to see the result immediately after it becomes available.

Provenance and Granularity. In acting as the single-point-of-entry for all functions, and observing both the input (accessible DEs), the list of actually accessed DEs obtained from the execution environment, and the function output, the Gatekeeper learns the provenance of each derived data product. The provenance is used primarily to understand if the results are accessible by a_i, as explained above. The provenance granularity depends on how the applications that register functions are built. For example, a *native* application built using Data Station APIs to access DEs can communicate precisely what DEs were used to produce the different outputs, and how. Data Station permits the execution of unmodified applications. We concentrate on the latter. Here, when the function's output DEs are a subset of the input DEs, the Gatekeeper checks for matching policies. For example, a search function that returns DEs that contain a keyword. When functions return results that are a combination of input DEs, then these newly created DEs, by definition, have no matching policies. For example, when a function trains a machine learning model off training datasets provided by different owners. The Gatekeeper relies on an additional mechanism to aid in handling these latter common cases.

3.2 Derived Data and Dependency Graph

When an intent refers to a DE uploaded by an owner, the Gatekeeper checks if there is a matching policy. But in many scenarios, functions require accessing DEs that are derived from other DEs [4]. For example, a search function may use an index, which may in turn be created by another function from a set of accessible DEs. The index is an intermediate DE. Because of that, no owner writes directly a policy for the index. When functions run on intermediate DEs, Data Station needs a mechanism to determine access to DEs and to facilitate the task of data owners when writing policies.

Function Dependency Graph. The dependency graph is a directed acyclic graph (DAG). Nodes represent functions, and directed edges represent dependencies between these functions in the form of DEs. For example, a *search* function depends on an *index* function, that itself creates the index from DEs. The dependency graph is created during the application registration process (Section 5). Owners understand the dependency between functions. When they write a policy to permit a function to execute on DEs they own, they are implicitly permitting any children of the function to access that DE as well. Hence, if they write a policy for the *search* function, they are also allowing *index* to access their DEs.

Policy Matching with Dependency Graph. To leverage the function dependency graph, during the policy matching process, the Gatekeeper must check whether there is an existing policy for the function, or any descendant. Without the dependency graph, if there is no matching policy, the Gatekeeper would only grant execution access to the function if the DE is in *enclave* mode. And this would later require brokering access to the function results, introducing unnecessary delays. Instead, the dependency graph permits skipping unnecessary handling of policies and vastly simplifies the task both for users, owners, and the Gatekeeper.

3.3 Auditable Computation: The Log

If Data Station keeps its promise, then owners are guaranteed that their DEs are accessed only in the way the policies they wrote permit. There are scenarios where auditors, compliance officers, and other third parties need access to the inner doings of Data Station. Because Data Station centralizes computation and data, they capture the provenance of every function invocation. This can be stored in a log and offered to operators as a source-of-truth record of what Data Station has done.

To be useful to operators, the auditability log must record *every* intent, policy match check, and DE result delivery that Data Station performs. It must permit data owners consulting the information that concerns DEs they own and it must permit data operators accessing this information when the participating agents have agreed to such arrangement. In other words: no computation can occur in Data Station without it being recorded in the log.

The Gatekeeper is the perfect candidate to manage this log because it already acts as a central actor checking every function invocation that aims to access DEs. Thus, the log, architecturally, resides inside the Gatekeeper. All writes to this log originate in the Gatekeeper, and all reads to the log go through the Gatekeeper, as with any other function invocation.

The log is a DE whose owner is Data Station. When a new owner joins Data Station, Data Station creates policies that permit owners to access the log and consult any activity that pertains to DEs they own. The owners can then inspect the log by invoking the corresponding function.

Log Entries. The log resides on disk and consists of a sequence of entries. Each entry consists of an agent id that corresponds to the caller that triggered the creation of the entry, and a payload. The payload can be of different types. It can indicate an intent was created, an intent-policy match, and a mismatch. And it also records what DEs are allowed outside the Station, including derived data products. Note that it is possible to trace what DEs contributed to any intermediate DE because the Gatekeeper keeps the provenance, which is itself materialized in this log. All history concerning DEs is stored in the log and this is the source of auditability.

Opening Access to Third Parties via Contracts. By default, the only agents who can inspect the log are owners. And they can only consult entries related to DEs they own. To permit access to third parties, all participating agents in Data Station must produce a contract. With a contract, Data Station creates a policy for the operator agent, who can then consult the log. A contract is a policy (a_o, r, l), where a_o is the id of the operator, r the read function on the log, and l the DE that refers to the log. A contract is different than a policy in that it must be signed by every participant. Without a signature per participant, Data Station does not create the policy, and the operator cannot consult the log. Data Station relies on public key cryptography to permit agents to sign contracts; these primitives are introduced in the next Section.

4 TRUSTWORTHY COMPUTATION

We introduce mechanisms used to protect Data Station's promise when running on untrusted infrastructure. We say Data Station runs in *near-zero-trust* mode when these mechanisms (introduced in Section 4.1) are activated. We present the encryption protocols used by Data Station in Section 4.2 and conclude by explaining how we deal with the log and the databases (Sections 4.3 and 4.4).

Threat Model. We assume that a curious operator gains access to Data Station infrastructure and can read disk and memory contents. To keep its promise, Data Station cannot leak any DE to this operator. Furthermore, Data Station cannot leak information from the database (that contains all agents, DEs, and policies) or auditable log. The adversary may gain access to a list of agent ids, but they should not be able to link those ids with any other information. We do not protect against denial-of-service attacks.

4.1 Near-Zero-Trust Principles

To ensure the confidentiality of every DE in Data Station, data is encrypted end-to-end, from the moment where it leaves the agent's infrastructure, and including while functions access that DE in memory, i.e., during processing. To ensure integrity, every message is signed with the private key of the agent from where it originates. To bootstrap trust, Data Station's node proves its identity to agents and attests that it runs the original Data Station software, and not a modified version, thus avoiding backdoor attacks.

Every shared DE is encrypted with an agent's specific symmetric key. It is then transmitted to Data Station over a secure channel with TLS [14]. Once in Data Station, it remains encrypted at rest. When a function needs to access the data, Data Station leverages secure hardware enclaves to maintain the data encrypted in memory, i.e., for processing, the DE is decrypted into encrypted memory. Functions may need to store intermediate results in the file system, such as temporal files. Because some of these intermediates may leak sensitive information, these intermediates are encrypted transparently to the application: no function can store plain data on disk when Data Station operates on *near-zero-trust* mode. We introduce secure hardware enclaves and the two key properties Data Station uses: encrypted memory and remote attestation.

4.1.1 A Primer on Secure Hardware Enclaves.

AMD's Secure Encrypted Virtualization (SEV) and Intel's Software Guard eXtensions (SGX) leverage specially-built hardware to isolate virtual machines (node) [29] and applications [25], respectively, within areas called *enclaves* to protect data leakage from even privileged users of the system. These technologies introduce important tradeoffs.

Compared to SEV that encrypts all of a node's working memory, SGX limits the total working memory to 128MB [24]. The upside is that with SGX users only need to trust the application that runs inside the enclave. In contrast, with SEV they must also trust the OS, which is ultimately responsible for ensuring memory pages are encrypted. Another downside of SGX is that applications need to be rewritten, as opposed to SEV, which accepts unmodified software. Since January'22, Intel has deprecated SGX [11]. This, in addition to the increased convenience of having all memory encrypted means we implement Data Station leveraging AMD's SEV.

Encrypted Memory. AMD's SEV guarantees confidentiality by encrypting all of the OS's writes to memory. Unauthorized users (including by the hypervisor in cloud contexts) cannot read data in plaintext, i.e., dumping the memory contents of the process (e.g., `cat /proc/[pid]/maps`) will show a cyphertext. With the more recent SEV-SNP (Secure Nested Paging) [50], the Trusted Computing Base (TCB) (the set of all hardware, firmware, and software that agents need to trust) only has two components: the AMD hardware and firmware, and the operating system image running in the node. All other components are untrusted, including the BIOS, hypervisor, other images (in the case of multi-tenant cloud scenarios), and external PCI devices. In summary, agents need only trust AMD's hardware is correctly implemented and that the OS image implements SEV correctly.

Attacks to Enclave Implementations and Limitations. Data Station operating in *near-zero-trust* inherits any vulnerabilities of the underlying SEV-SNP implementation. Solving SEV's implementation specific bugs [36, 37] is outside the scope of this paper, and AMD actively works to mitigate them at the time of writing. SEV enclaves protect the confidentiality and integrity of data in main memory, but not of data living on external devices, such as disks or GPUs. Data Station offers protection for data on disk as well as its transfer to memory, but it does not support computation on the GPU. Another limitation of enclaves is their reduced performance: we show in the evaluation section that this overhead does not affect application runtime significantly. Finally, every major vendor provides a secure enclave technology, but the specific security guarantees of SEV that make it a good fit for Data Station are, as of May'22, only provided by AMD and available in the Google Cloud.

4.1.2 Bootstrapping Trust with Remote Attestation.

To convince agents that the infrastructure running Data Station has SEV enabled and that the software running is indeed Data Station software, the platform uses remote attestation, as provided by AMD's hardware. This permits agents request a report from the node that contains unique identifying information. Furthermore, the agents can trust the node is running a version of Data Station with a correct implementation of the software, including the Gatekeeper, as opposed to an adversarial modified version that includes a backdoor. If the software restarts, the entire remote attestation process repeats to avoid an attacker swapping software versions.

4.2 e2e Encryption Protocol and Key Manager

The end-to-end encryption protocol must ensure all DEs are always encrypted, including during processing. At the same time, it must permit agents with a matching policy to access the DEs. Two components are primarily responsible for achieving this goal, the *Protocol Coordinator* that uses mostly standard public-key cryptography, and the *Volatile Key Manager* that is the mechanism used to protect cryptographic keys. We explain both next.

Preliminaries. Every agent and Data Station have a public and private key. In addition, every agent has a symmetric key that they share with Data Station. Data is encrypted and signed (to prove identity) at origin and remains encrypted throughout the lifecycle.

Protocol Coordinator. To process a DE, Data Station first decrypts it in memory by using the symmetric key shared by the DE's owner.

Because memory in the enclave is encrypted, the data remains protected from external observers. To transmit a DE (e.g., to an agent with a matching policy) Data Station first decrypts the DE in memory and re-encrypts it with the symmetric key of the receiving agent. Again, the DE remains encrypted at all times because this processing takes place in-memory.

Volatile Key Managemer. Data Station possesses two classes of sensitive information: i) the symmetric keys used by agents to encrypt the DEs; ii) its own private key. If any of these keys is compromised, the whole system's guarantees fall apart. To protect these keys, they are stored in a volatile key manager that resides in-memory, hence encrypted. This is a simple way of maintaining the keys secure while Data Station is running. However, if Data Station restarts (e.g., failure, maintenance), the keys will vanish from the key manager, so a strategy to recover them is needed. Some enclave technology, such as Intel's SGX supports *sealing* data, which means encrypting in-memory data to disk with an enclave-specific key. AMD SEV does not support *sealing* [53]. Hence, Data Station relies on agents resending their encryption keys to recover the Key Manager state when necessary.

Derived Data Products. Data Station is responsible for encrypting derived data products before storing them on disk. Notice that there is no need to use any specific key. Unlike with shared DEs, derived data products exist because owners let Data Station invoke functions on them. It follows Data Station can select what key to use to maintain their encryption on disk. Data Station uses the calling agent's symmetric key to avoid one round of re-encryption if the calling agent is given permission to access the DE.

4.3 Audit Log Management

If the audit log was stored in plain text, an attacker who gains access to the node would learn what functions were executed on what DEs and by whom, and thus could learn about existing policies. Although this may not be critical in some applications, Data Station protects against this risk by maintaining the log encrypted on disk. The log is a sequence of $\langle a_i, E_{k_u}(l) \rangle$ entries. The agent ID, a_i, is in plain text, but it is only meaningful with access to the database.

Only owners can see log entries that involve DEs that they own, so naturally one may think that these entries are encrypted with the owner's key. However, an entry may involve many DEs, e.g., functions that require accessing collections of DEs, and encrypting the entry with each DE's owner's key is inefficient when the number of DEs is large. Instead, Data Station encrypts the log entries with the symmetric key, k_u, of the data user whose function call triggered these entries. Because only Data Station can access the audit log and each entry is generated by exactly one agent, this solution is more efficient.

4.4 Database Management

An attacker who gains access to the infrastructure where the database is hosted learns all policies, DE locations (but not access, as these are encrypted), and information about any agents registered with the platform. To protect against this, the database remains in-memory, and hence, encrypted with SEV. Then, even when an attacker gains access to the physical machine, the contents of the database are protected. The challenge then is how to deal with failures and reboots of Data Station.

To solve this problem, Data Station uses an encrypted write-ahead-log (EWAL) that is independent of, and external to any WAL used internally by the database system. All updates to the database are first stored encrypted in the EWAL. The entries are encrypted by using the symmetric key of the agent who generated the update. Each entry is stored along with the agent id that caused the update. The assignment of IDs to agents takes place outside the database so these can be incorporated in the EWAL log entries. After restarting, the database can be recovered from the log, as in the traditional recovery protocols of relational databases [41]. To deal with a growing EWAL, the database can be checkpointed to disk periodically. Ahead of storing the checkpoint on disk, this must also be encrypted in memory (see below). After a restart, decrypting the EWAL entries requires collecting keys from the agents first. After the EWAL is replayed, Data Station is considered recovered. With a recovered database, Data Station can recover, in turn, the audit log.

Encrypting Database Checkpoint without a Sealing Mechanism. The database checkpoint must be encrypted ahead of being stored on disk. With sealing, Data Station would use the enclave key to encrypt the checkpoint. Without a sealing mechanism, we resort to a different solution. The baseline solution is to select one of the symmetric keys of agents to encrypt the checkpoint. On restart, agents will resend their keys to Data Station, which attempts to decrypt the checkpoint with each newly received key until it succeeds. Because operation cannot be resumed until all agents have resent their keys, this process does not introduce additional delays. To increase reliability, i.e., in case the agent whose key encrypted the checkpoint does not reconnect to Data Station, the checkpoint can be encrypted with m agents' keys instead, and then as soon as one key decrypts the database Data Station becomes operational.

4.5 Other Protections and Limitations

Potential adversaries that gain control of the infrastructure cannot modify the contents of the database because it is encrypted. Hence, they cannot create policies, agents, or change the sharing mode of existing DEs. Furthermore, these attacks to the integrity of the memory contents will be discovered when *SEV-SNP* becomes available. All interactions with Data Station are mediated via the Protocol Coordinator, who among other things, handles authentication.

Although the EWAL and audit log are encrypted on disk, an attacker could potentially perturb these, e.g., by adding random bytes. Data Station is not protected against such denial of service attacks. This limitation of the implementation could be addressed by extensions to support a replication service.

5 EXECUTION ENVIRONMENT

Requirements. The execution environment comprises all the functionality and components of Data Station that permit the execution of applications. In designing Data Station, we wanted to permit existing applications execute unmodified to ease their deployment. The requirements for the execution environment are:

- Developers *register* existing applications with Data Station without making changes to the application's implementation. They provide a simple *connector* that indicates how to invoke the application functionality.
- Data Station provides the application with the necessary resources to execute and serve function invocations while guaranteeing the application is isolated from other applications and from data it cannot access.
- During execution, the DEs accessed by the application must be recorded by the execution environment and sent to the Gatekeeper after the function finishes.
- In the *near-zero-trust* mode, data is encrypted on disk. The execution environment ensures applications access decrypted data transparently so they do not break during execution.

5.1 Design

Function Registration. Application developers *register* functions with the Gatekeeper via a *connector*. The connector contains the functions exposed to Data Station and the dependencies between these functions. This is all the information Data Station needs to build the *function dependency graph*. Upon system initialization, Data Station loads all connectors from developers to register the functions with the gatekeeper. Each function in the connector is in charge of invoking the functionality from the application.

Resource Management and Isolation. When the Gatekeeper grants execution permission to a function, the function is instantiated in an isolated process with restricted and controlled access to the system's resources.

Interceptor Middleware. Data Station starts an Interceptor upon initialization. The Interceptor knows all processes started by the Gatekeeper and the locations such processes can access. It intercepts all I/O calls from the function's isolated process to storage. The Gatekeeper and the Interceptor middleware work in a client-server manner as follows:

- The Gatekeeper passes the list of accessible DEs to the Interceptor through the function's execution environment. The Interceptor makes sure that only those accessible DEs are visible to the function by filtering out those that are not.
- The Interceptor records all DEs actually accessed by the function and sends them back to the Gatekeeper, again through the function's execution environment.
- When operating in *near-zero-trust* mode, data is encrypted on disk, and no function side effects (such as temp files) should be stored in plain text. Whenever a function invokes a read operation, the Interceptor decrypts the data on-demand; analogously, it encrypts writes using the data owner's symmetric key.

Last, the Gatekeeper obtains from the Interceptor the list of DEs the function actually accessed and proceeds as explained earlier.

5.2 Implementation

The executor environment operates on a local file system. Extending it to other settings is beyond the scope of this paper.

The Gatekeeper creates isolated processes using Docker containers. Each container effectively creates a jail (in the chroot sense) that limits the resources accessible by functions. The container is created so that all paths the function can access are intercepted by the Interceptor. Furthermore, instantiating functions in Docker containers permit easy management of the resources available.

We implement the Interceptor middleware by using FUSE [54], with the libfuse userspace library [2] used to create file systems in userspace. During Data Station's system initialization, the Interceptor mounts Data Station's storage to a specified mountpoint. All subsequent function invocations will access DEs through the mountpoint, so that all I/O operations can be intercepted.

Using FUSE has one key advantage—since the filesystem is created in user space, we do not need to modify the kernel code to intercept the I/O calls and perform additional operations. Avoiding kernel modification is necessary to reduce the size of the trusted computing base. The FUSE filesystem offers a flexible way to interact with Data Station without compromising its security guarantees.

To support concurrent function invocations, the Interceptor works as a server that accepts requests from the gatekeeper, which starts an isolated process per function invocation. The Interceptor keeps track of the corresponding DEs accessed by each running function. This is achieved by associating each data access with the identity of the execution environment that attempts to access the data. Similarly, the list of accessible DEs (and the corresponding symmetric keys if operating in near-zero-trust mode) are also associated with the identity of an execution environment. The Gatekeeper and Interceptor must remain connected to ensure the right process contextual information is shared.

6 EVALUATION

We present the evaluation results to answer two questions:

RQ1: What applications does Data Station enable? We implement two types of applications in Data Station: machine learning applications and a file-sharing application. We find that Data Station achieve much better quantitative results and present a number of invaluable qualitative advantages.

RQ2: Do Data Station's design decisions lead to a practical system? We study the overheads introduced by the need to provide trustworthiness and the use of SEV. Our aim is to determine whether such overheads pose a runtime bottleneck to applications. We study each component in Data Station as well as the overhead introduced by SEV. We show that overheads are negligible even when running in *near-zero-trust* mode.

Outline. We divide our evaluation into three subsections, the first two address **RQ1** and present the quantitative and qualitative results for two applications. The last presents the characterization of overheads of Data Station, thus addressing **RQ2**.

6.1 Machine Learning Consortium

We consider a scenario where $N = 8$ agents want to pool their individual datasets, D_i, to train a more powerful machine learning model but without allowing other agents to see their raw data. Only access to inferences on the jointly trained model is permitted. When using Data Station (in *near-zero-trust* mode), each agent registers their data with the platform because they know their data will be protected end-to-end. We also consider a federated learning scenario, where agents' data never leaves their own machine and

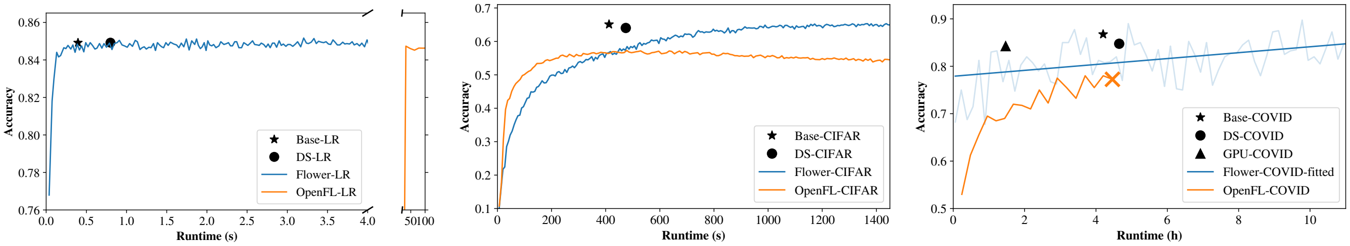

Figure 2: Accuracy and runtime of the four baselines on adult income (left), CIFAR-10 (center), and COVIDx (right).

the model is trained in a distributed manner. The federated learning setting corresponds to the closest setting we can deploy to achieve the desired goal, and we compare federated learning with Data Station from a qualitative perspective later in this section.

Baselines. We consider the following baselines:

- Base. This is a centralized untrusted server training the model. It serves as a reference to compare with other baselines.
- DS. Data Station running in *near-zero-trust* mode.
- Flower [10]. A state-of-the-art federated learning framework implementing the FederatedAveraging aggregation algorithm [40]. We sample from 2 clients after each round to accelerate convergence and as recommended in the documentation. We deploy Flower in a distributed setting, with each client/agent accessing its own node.
- OpenFL [47]. A state-of-the-art federated learning framework that implements the same aggregation algorithm as Flower but does not sample per epoch. Its support for distributed computing is not mature; we deploy all clients in a single node.

When exploring federated learning baselines we considered others such as PySyft [66] and FATE [1]. We choose Flower and OpenFL because they implement state-of-the-art federated aggregation algorithms and are mature: i) they are actively maintained; ii) their documentation is complete; iii) the examples in their tutorials work.

Machine Learning tasks. We consider 3 machine learning tasks:

- Logistic Regression on the Income dataset [56]. The task is to predict whose salary is <$50K. In the federated learning baselines, each agent has access to a 4.5K-sample even split of the dataset.
- Computer vision on the CIFAR-10 dataset [32]. Each agent has access to 6,250 samples.
- Computer vision on the COVIDx CXR-3 dataset [61]. Each agent has access to 3810 chest-ray images. The task is to predict whether the patient has COVID-19. The total size of the dataset is 14GB and the model used is neural network.

Experimental Setup. We use n2d-highmem-8 instances (8 virtual CPUs and 64GB of RAM) from Google Compute Engine. In particular, we use 8+1 (clients+server) such nodes for Flower; in the case of OpenFL, we run all clients and server in a single node because the framework does not work well in a distributed setting. We run Base and DS in a single node with the same specs as the other nodes, but enabling SEV when running DS to support *near-zero-trust* mode. We use the same logic for pre-processing, training, inference, and evaluation across baselines, with one exception. OpenFL on income is implemented using a Keras model (instead of sklearn which we use in the others) becasue of the limited support of the framework for other libraries.

Metrics and Baselines. We evaluate the time it takes to train the model and the maximum accuracy achieved in the four baselines.

Results. Fig. 2a, Fig. 2b, Fig. 2c show the results of the experiment for the three tasks. The performance of Flower and OpenFL is shown as a convergence line. We fit a line to increase readibility when the underlying convergence behavior is spiky. We use dots that indicate when convergence is achieved for Base and DS. We observe several trends.

First, on the same amount of time, centralized training (Base and DS) achieve higher accuracy than the federated learning baselines. Second, the federated learning baselines take 3x more time (e.g., 11 hours vs 4 hours in Fig. 2c) to achieve a similar accuracy than DS. This is true for the more complex deep network models: CIFAR (Fig. 2b) and COVIDx (Fig. 2c). In the much simpler, logistic regression model used for the income experiment, the centralized approaches and Flower perform similarly well; OpenFL takes a bit longer to produce results but otherwise achieves a similar accuracy.

Second, both federated learning frameworks perform similarly. OpenFL is less stable and runs out of memory in the COVIDx experiment (the X in Fig. 2c). Note that we run the federated learning baselines in their *best-case-scenario*, where each client has a random sample of the training data. When this is not true (common in practice) the efficiency of federated learning reduces.

Third, the runtime overhead of DS with respect to Base is minimal across all datasets, despite running in *near-zero-trust* and maintaining clients' data encrypted end-to-end. However, DS running in *near-zero-trust* cannot run computation on GPUs, so when doing so brings performance benefits, DS leaves those on the table. We demonstrate that in Fig. 2c, by including Base running on a GPU and showing the performance difference with DS.

The results show the advantages in accuracy and runtime of centralizing data and compute, and the low overhead of Data Station compared to off-the-shelf (non-trusted) model training.

Qualitative Analysis. There are various important qualitative differences between Data Station and federated learning:

Compatibility. Federated learning supports only machine learning models that can be merged. Data Station has no such limitation. Similarly, existing applications must be modified and adapted to the federated learning framework. Data Station only requires writing a simple connector to expose the functions.

Security and Leakages. Many federated learning frameworks leak weight updates [27], which can be leveraged to reconstruct part of the data, thus breaking the promised security guarantees. A solution

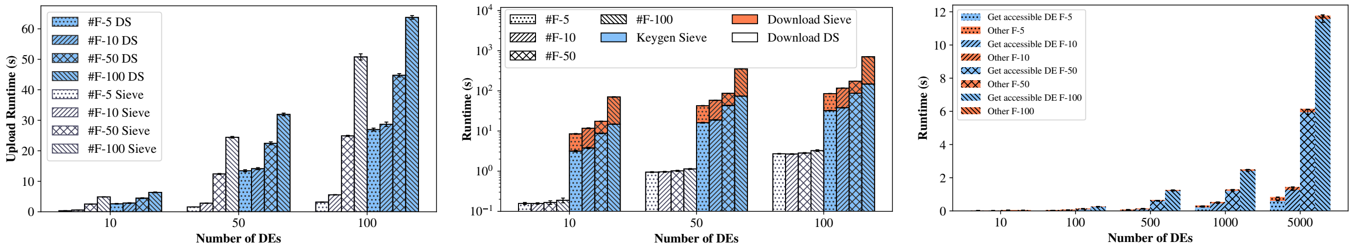

Figure 3: Left (Center): Upload (Download) time of Data Station and Sieve. Right: Data Station overhead on data users

is combining federated learning with differential privacy [59]. This requires further modifications to applications and algorithms, and results in further performance reduction.

Performance Differences. Many federated learning algorithms do not guarantee the same performance as centralized implementations. We demonstrated this quantitatively above.

Scalability. While (in principle) federated learning scales with the number of data contributors, the current design of Data Station would need to adapt to support multi-machine setting. We do not anticipate severe barriers in achieving that goal.

6.2 Secure Data Sharing

In this scenario, owners store data in a third-party server and selectively let other agents access this data by writing policies. In this scenario, we execute Data Station in *near-zero-trust* mode, and so it is subject to overheads from the enclave and the encryption protocol. We want to understand whether these overheads lead to an impractically slow platform. To answer that question, we measure the runtime and compare it with Sieve's [58].

A Primer on Sieve. Sieve enables cryptographically secure data sharing via an untrusted intermediary. It uses symmetric keys for encrypting files, like Data Station. It encrypts metadata for each file with attribute-based-encryption (ABE) [22] and this allows it to generate decryption keys that only work with selected attributes. Finally, it relies on homomorphic encryption [20] for revoking access to datasets. Unlike Data Station, Sieve is exclusively designed for file sharing. Sieve is the only approach we identified that simultaneously addresses the secure data sharing problem, is open source, and uses cryptographic techniques to build trust with data owners. The role of Sieve in our evaluation is to provide a reference performance to facilitate interpreting the performance of Data Station when running in *near-zero-trust* mode.

Experimental Setup. Sieve can generate decryption keys that only decrypt data previously encrypted with specific attributes. We use one Sieve attribute per registered function inside Data Station. Assume there are n DEs and m functions. Sieve encrypts each dataset with a symmetric key and the metadata (the functions) with ABE. This lets a user who wants to invoke a function $f \in m$ download a dataset in Sieve. Data Station runs in *near-zero-trust* mode and implements a download application. download takes a DE as input parameter and, if permitted by the gatekeeper, sends the DE to the user. This replicates the Sieve setup with the same functionality, thus letting us compare both approaches.

Metrics. We evaluate the end-to-end performance of data sharing by measuring the time for the data owner to upload datasets and the time for the data user to download datasets. In Data Station, we create $n * m$ policies to indicate that a user can invoke any function on any DE. We use files each consisting of 10KB random byte strings. We show average runtime over 20 runs.

Upload Results. Fig. 3a shows the results for uploading data with Data Station and Sieve when changing the number of DEs (x axis) and the number of registered functions (different bar styles). First, when the number of registered functions (#F-m) is small, Data Station is an order of magnitude faster than Sieve. When the number of functions increases to 50, Data Station is 80% faster than Sieve, and at 100 functions Data Station is still 25% faster. Second, the scaling behavior in both Data Station and Sieve is similar and depends on the number of DEs and functions. However, what is crucially important is that, in Data Station, we are measuring the *worst case scenario* where every DE can be downloaded by any function. Note that, when this is not the case, Data Station's overhead will reduce (with the number of DEs and actual functions allowed), while in the case of Sieve it will remain constant because it still needs to encrypt a value for the attribute. Finally, in Data Station the dominating cost is creating policies that requires writing to the EWAL as it operates in *near-zero-trust*. In Sieve, the symmetric encryption of the file varies only with the number of DEs uploaded. The most significant Sieve overhead is the ABE encryption time, accounting for over 95% of the upload time among all DEs and attributes. Data Station outperforms Sieve's even in the worst-case scenario of allowing every function to execute on every DE.

Download Results. Fig. 3b shows the results for downloading data when changing the number of DEs (x axis) and the number of registered functions (different bar styles). After a user calls download and the Gatekeeper checks they have access the function reads the encrypted DE into memory, decrypts it with the owner's symmetric key, re-encrypts it with the user's symmetric key and sends it to the user. Sieve's higher runtime stems from the use of ABE. Sieve's owner has to generate a decryption key (Keygen) via ABE for the necessary attributes, shown in the bottom bar of Fig. 3b. The generated key is sent to the user, who uses it to decrypt the data; this decryption accounts for over half of the time spent downloading. Although Sieve's design allows the key generation to execute only once for a user to gain access to the data, the decryption and download costs must still occur. Data Station thus provides two benefits over Sieve: i) owner needs no interaction with the user downloads; ii) runtime is between 1 and 2 orders of magnitude lower.

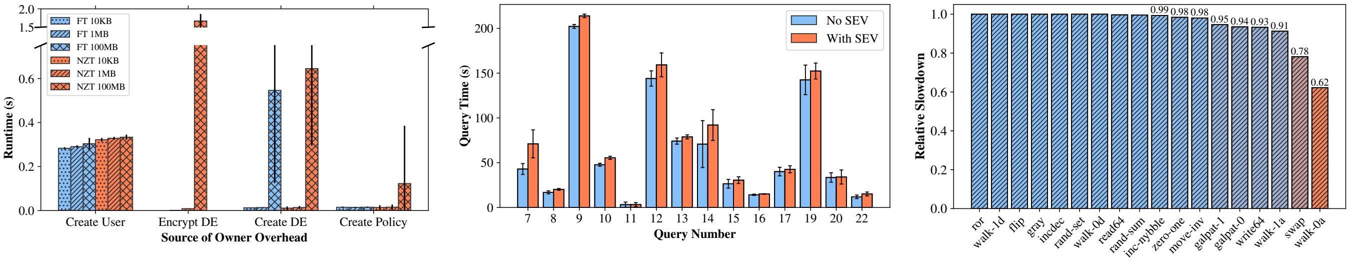

Figure 4: Left: Overhead for Data Owners. Center: SEV Overhead on TPC-H Benchmark. Right: stress-ng Benchmark.

Qualitative Analysis. We complement the quantitative results with important qualitative differences between approaches.

Trust Model. Unlike Data Station, Sieve does not hide file metadata (i.e., policies) from the infrastructure provider. Thus, a curious provider will learn the functions available in the platform and who has access to them, even though they cannot read the raw data. Data Station protects this information by keeping the database, EWAL, and audit log encrypted at all times. A curious provider may learn agent IDs but cannot associate them with users.

Delegated Computation. Sieve does not support delegated computation, it acts as a storage provider. The implication is that once a user is given access to a DE, the owner in Sieve must connect directly to the user. This is unlike Data Station, where we can provide a function to perform this delegated computation, in this case the simple download function, but in general, more complex functions as we saw in the machine learning application above.

Revoking Access to Data. In Data Station, revoking access to data translates to modifying a policy, which is a quick operation, and can be done selectively for a given user. In contrast, in Sieve this involves changing the attributes for the previously uploaded data and then re-encrypting the file on the storage provider. Because the file cannot be decrypted first, Sieve uses a clever technique to re-encrypt it without decryption using homomorphic encryption. This has several implications. First, the newly generated key must be re-sent to those users who still have access to the file. Second, the revocation process itself is much slower than in Data Station because of the aforementioned techniques.

Sieve is a well designed and engineered system. We believe the results are representative of solutions fully based on cryptographic techniques and thus validate Data Station's design.

6.3 Data Station Performance Analysis

The previous two experiments demonstrate that the overhead Data Station introduces is small compared to application time. Here, we use a noop function to understand Data Station's *fixed* overheads for users and owners. We conclude by measuring the overhead AMD SEV introduces in Section 6.3.3.

6.3.1 Overhead for Data Users. After a user invokes a function, the potential sources of overhead are: i) obtain user's ID from the database; ii) obtain accessible DEs from policy broker; iii) initialize execution environment; iv) collect list of actually accessed DEs after function invocation; and v) log all activity in the audit log.

Overhead Factors. In the noop operation, no DE is ever accessed so no DE characteristics (e.g., data type) affect the overhead. The DE size has an effect when they need to be decrypted (when operating in near-zero-trust mode), but we already studied such an effect in Section 6.2. Here, we concentrate instead on the 5 steps above. Of those, 'obtaining the list of accessible DEs' dominates the overhead, and there are two factors that affect such overhead: the number of registered DEs and the number of functions registered. Consequently, we show results varying these two factors.

Experimental Setup. When varying the number of functions registered, we create a policy for each pair of DE and function, and store these policies in the database. We report the average end to end runtime over five runs. Neither DE size nor type affect overhead; we use 1MB DEs representing text files.

Results. Fig. 3c shows the runtime per function invocation when changing the number of DEs (x axis) and for different numbers of functions. We report the runtime for obtaining the list of accessible DEs, and we group the other sources of overhead together and refer to them as Other. The overhead for getting the accessible DEs scales linearly with the number of policies. With 5000 DEs and 100 functions (i.e., half a million policies) the overhead is only 12 seconds. Contrast that with the multiple minute (and hours) overhead of training a machine learning application such as that in the previous section, which has only a few registered functions. We conclude that the overhead for data users is negligible for applications with runtimes larger than a minute.

6.3.2 Overhead for Data Owners. The sources of overhead for data owners are: i) registering with the platform; ii) (only in near-zero-trust mode) encrypting the DEs to upload; iii) uploading DEs; and iv) creating policies. Unlike before, the size of DE matters because of sources ii) and iii), so we vary the DE size in this experiment.

Results. Figure 4a shows the sources of overhead in the x axis. We show results for DEs of different sizes to expose the overheads introduced by encryption (Encrypt DE) and uploading the DE (Create DE). We report average over 100 runs. The overhead of Create User, Create DE and Create Policy is always slightly higher in *near-zero-trust* mode due to the use of SEV and writing to the EWAL, but remain sub-second. The largest source of overhead is encrypting DEs (only in *near-zero-trust* mode), which depends on DE size.

6.3.3 AMD SEV's Performance Overhead. We measure the overhead introduced by AMD's SEV using the TPC-H benchmark, as a representative data-intensive workload familiar to the reader, and stress-ng [31], a tool to stress computer systems and used in previous work to understand enclave's performance [21]. We run TPC-H on DuckDB [46] over a 100GB database with and without SEV. Fig. 4b shows the average runtime for both baselines after 10 runs of each query. The results show that SEV indeed introduces overhead; most noticeable in query 7 that writes a large amount of disk-resident data into (encrypted) memory. However, the overhead is modest given the security guarantees gained. The results cement the advantages of modern secure hardware enclaves and the opportunities that they open.

stress-ng [31] uses *stressors* to understand system performance. Fig. 4c shows the relative slowdown of a stressor with SEV active, measuring the number of operations completed per unit of time (slowdown = $\frac{\text{SEV ops done}}{\text{no SEV ops done}}$). The stressors that introduce the largest overhead (e.g., walk-0a, swap, walk-1a) correspond to those performing random-access, memory-intensive tasks. The walk-Xa stressors force reads from physical memory, which explains the larger overhead, and swap exchanges the contents of two different memory locations.

7 RELATED WORK

Data Station is the first data escrow system that concentrates in offering delegated, trustworthy, and auditable computation. It builds on many existing lines of research that we explain below.

Hippocratic Databases [3]. Data Station is related to the Hippocratic databases vision. There are important similarities and differences. Crucially, in both papers access control is defined around a *purpose*, which is specified in Data Station with a policy referencing a function. In this way, both vision and system chase a contextual integrity [43] view of privacy more so than one based on access control [60]. Another similarity is the need for auditable computation, via the audit log in Data Station and the concept of audit trails in Hippocratic databases. The differences are also important. Hippocratic databases are envisioned as a database system that takes care of privacy. Data Station is a data escrow system for implementing other applications, including RDBMS but also ML and other analytics-based functionality. As Data Station decouples computation and data, a crucial element is the need to build trust with users and owners alike.

Data Enclaves store restricted-use data, e.g., data subject to privacy and regulation constraints [18]. The enclave ensures that the hosted data is protected while permitting users to execute certain pre-determined computations, often with review prior to data release [33, 62]. They are commonly found in research organizations, where they are built with the intention of facilitating data-driven research, e.g., ICPSR in the social sciences [26], and NORC [44]. These enclaves are the result of long sustained efforts. Data Station is geared towards easing the creation of data enclaves, including to share data among organizations.

Multi-Party Computation, Homomorphic Encryption, Federated Learning, and Differential Privacy. There is a growing class of systems designed to permit collaborative analytics on restricted-use datasets. Shrinkwrap [7] and Saqe [8] permit the execution of SQL queries over data from multiple organizations, while ensuring differentially private results that do not disclose the identity of any participant. Conclave [57], Cerebro [64], and Secrecy [38] rely on multi-party computation to achieve a similar goal. Alternative techniques such as homomorphic encryption permit run computation directly on encrypted data. These technologies concentrate on providing trustworthy data processing. Unlike existing solutions and technologies, Data Station is designed to tackle the three requirements presented above, which no other solution achieves.

Confidential Computing initiatives have been promoted by cloud vendors to build trust with customers who store their data in the cloud. Azure Always Encrypted [5] lets users run computation on data they own on infrastructure they do not own (i.e., the cloud). Similarly, research approaches such as Haven [9], SCONE [6], Keystone [35], Ryoan [23], VC3 [49], protect data processing when the data is hosted in the cloud. Many of them focus on protecting databases [19, 52, 63, 65]. Some recent work has proposed infrastructure to facilitate running SQL queries across data from multiple parties [13]. All these solutions leverage secure hardware enclaves, like Data Station. But unlike Data Station, none of them enable delegated, trustworthy, and auditable computation on data owned by multiple parties.

Auditability. Many have noted the importance of building audit logs, in IoT environments [45], using blockchain technology [39, 48, 51] to make the log tamper-proof, and even building databases on top of that abstraction [16]. Similarly, there is work proposing ways of reasoning about access policies in the context of databases [4]. The techniques used in these approaches are complementary to the ones we use in the audit log in Data Station. Unlike these approaches, Data Station leverages the centralization of data and compute to simplify the problem.

8 CONCLUSIONS

The increasing number of scenarios where organizations benefit from pooling and sharing data calls for technical solutions to ease the task of forming data-sharing consortia. We have presented Data Station, a data escrow system that implements delegated, trustworthy, and auditable computation with the aim of facilitating owners and users to share and benefit from each others' data. Data Station supports unmodified applications and thus a wide range of applications. We presented mechanisms to generate trust from owners and users based on the use of secure hardware enclaves and cryptographic protocols. The evaluation results demonstrate the feasibility of the approach when compared to strong baselines for other applications, including machine learning training consortia and secure data sharing. We study the overheads and demonstrate they are negligible compared to application runtime. Beyond the quantitative differences, we highlight important qualitative advantages of the data escrow design.

ACKNOWLEDGMENTS

This work was supported by the NSF Convergence Accelerator (Award Number #2040718). We thank Frank Wang for helping us set up Sieve and Christian Goettel for helping us benchmark SEV. We also thank the Chameleon cloud [30] for providing computational resources that helped with the evaluation and testing of our system, and the anonymous reviewers for providing valuable feedback that helped us improve the paper.

REFERENCES

[1] [n.d.]. FATE. https://fate.fedai.org/ Online; accessed 29 May 2022.

[2] [n.d.]. Python-fuse interface to libfuse. https://github.com/libfuse/python-fuse. Online; accessed 29 May 2022.

[3] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. 2002. Hippocratic databases. In *28th International Conference on Very Large Databases*. Elsevier, 143–154.

[4] Yael Amsterdamer and Osnat Drien. 2020. Towards Fine-Grained Data Access Control Through Active Peer Probing.. In *EDBT*. 403–406.

[5] Panagiotis Antonopoulos, Arvind Arasu, Kunal D Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, et al. 2020. Azure SQL database always encrypted. In *ACM SIGMOD International Conference on Management of Data*. 1511–1525.

[6] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. 2016. SCONE: Secure Linux containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 689–703.

[7] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. 2018. Shrinkwrap: Differentially-private query processing in private data federations. *Proceedings of the VLDB Endowment* 12, 3 (2018), 307–320.

[8] Johes Bater, Yongjoo Park, Xi He, Xiao Wang, and Jennie Rogers. 2020. Saqe: practical privacy-preserving approximate query processing for data federations. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2691–2705.

[9] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst.* 33, 3, Article 8 (aug 2015), 26 pages. https://doi.org/10.1145/2799647

[10] Daniel J Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Titouan Parcollet, Pedro PB de Gusmão, and Nicholas D Lane. 2020. Flower: A friendly federated learning research framework. *arXiv preprint arXiv:2007.14390* (2020).

[11] CCG CCE Tech Pubs - Intel Corp. 2022. 12th Generation Intel® Core™ Processors — Datasheet, Volume 1 of 2. https://www.intel.com/content/www/us/en/products/docs/processors/core/core-technical-resources.html. Online; accessed 28 February 2022.

[12] Chris Clifton, Murat Kantarcioğlu, AnHai Doan, Gunther Schadow, Jaideep Vaidya, Ahmed Elmagarmid, and Dan Suciu. 2004. Privacy-preserving data integration and sharing. In *9th ACM SIGMOD workshop on Research Issues in Data Mining and Knowledge Discovery*. 19–26.

[13] Ankur Dave, Chester Leung, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2020. Oblivious coopetitive analytics using hardware enclaves. In *15th European Conference on Computer Systems*. 1–17.

[14] Tim Dierks and Eric Rescorla. 2008. The transport layer security (TLS) protocol version 1.2. (2008).

[15] Peter F Edemekong, Pavan Annamaraju, and Micelle J Haydel. 2018. Health insurance portability and accountability act. (2018).

[16] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. 2019. BlockchainDB: A shared database on blockchains. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1597–1609.

[17] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. 2018. Aurum: A data discovery system. In *IEEE 34th International Conference on Data Engineering*. IEEE, 1001–1012.

[18] Ian Foster. 2018. Research infrastructure for the safe analysis of sensitive data. *The Annals of the American Academy of Political and Social Science* 675, 1 (2018), 102–120.

[19] Benny Fuhry, HA Jayanth Jain, and Florian Kerschbaum. 2021. Encdbdb: Searchable encrypted, fast, compressed, in-memory database using enclaves. In *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 438–450.

[20] Craig Gentry. 2009. *A fully homomorphic encryption scheme*. Ph.D. Dissertation. Stanford University. https://crypto.stanford.edu/craig/craig-thesis.pdf.

[21] Christian Göttel, Rafael Pires, Isabelly Rocha, Sébastien Vaucher, Pascal Felber, Marcelo Pasin, and Valerio Schiavoni. 2018. Security, performance and energy trade-offs of hardware-assisted memory protection mechanisms. In *IEEE 37th Symposium on Reliable Distributed Systems*. IEEE, 133–142.

[22] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. 2006. Attribute-based encryption for fine-grained access control of encrypted data. In *13th ACM Conference on Computer and Communications Security*. 89–98.

[23] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2018. Ryoan: A distributed sandbox for untrusted computation on secret data. *ACM Transactions on Computer Systems* 35, 4 (2018), 1–32.

[24] Intel Corporation. 2017. Enclave Memory Measurement Tool for Intel® Software Guard Extensions (Intel® SGX) Enclaves. https://www.intel.com/content/dam/develop/external/us/en/documents/enclave-measurement-tool-intel-sgx-737361.pdf. Online; accessed 24 February 2022.

[25] Intel Corporation. 2021. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D: System Programming Guide, Part 4. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. Online; accessed 23 February 2022.

[26] Inter-university Consortium for Political and Social Research. 2022. ICPSR Data Enclaves. https://www.icpsr.umich.edu/web/pages/ICPSR/access/restricted/enclave.html. Online; accessed 18 February 2022.

[27] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. 2021. Advances and open problems in federated learning. *Foundations and Trends® in Machine Learning* 14, 1–2 (2021), 1–210.

[28] Poul-Henning Kamp and Robert NM Watson. 2000. Jails: Confining the omnipotent root. In *2nd International SANE Conference*, Vol. 43. 116.

[29] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD memory encryption. *White paper* (2016).

[30] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association.

[31] Colin Ian King. [n.d.]. stress-ng. https://github.com/ColinIanKing/stress-ng Online; accessed 29 May 2022.

[32] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[33] Julia Lane, Pascal Heus, and Tim Mulcahy. 2008. Data Access in a Cyber World: Making Use of Cyberinfrastructure. *Transactions on Data Privacy* 1, 1 (2008), 2–16.

[34] Federated Learning. 2017. Collaborative machine learning without centralized training data. *Publication date: Thursday, April* 6 (2017).

[35] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An open framework for architecting trusted execution environments. In *15th European Conference on Computer Systems*. 1–16.

[36] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. 2021. CROSSLINE: Breaking" Security-by-Crash" based Memory Isolation in AMD SEV. In *ACM SIGSAC Conference on Computer and Communications Security*. 2937–2950.

[37] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. {CIPHERLEAKS}: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *30th USENIX Security Symposium (USENIX Security 21)*. 717–732.

[38] John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia. 2021. Secrecy: Secure collaborative analytics on secret-shared data. *arXiv preprint arXiv:2102.01048* (2021).

[39] Sujaya Maiyya, Victor Zakhary, Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. Database and distributed computing foundations of blockchains. In *International Conference on Management of Data*. 2036–2041.

[40] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. 2016. Communication-Efficient Learning of Deep Networks from Decentralized Data. (2016). https://doi.org/10.48550/ARXIV.1602.05629

[41] Chandrasekaran Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)* 17, 1 (1992), 94–162.

[42] Nightingale Open Science. 2022. https://www.nightingalescience.org/. Online; accessed 25 February 2022.

[43] Helen Nissenbaum. 2004. Privacy as contextual integrity. *Wash. L. Rev.* 79 (2004), 119.

[44] NORC. 2022. NORC Data Enclave. https://www.norc.org/Research/Capabilities/Pages/data-enclave.aspx. Online; accessed 18 February 2022.

[45] Nisha Panwar, Shantanu Sharma, Guoxi Wang, Sharad Mehrotra, Nalini Venkatasubramanian, Mamadou H Diallo, and Ardalan Amiri Sani. 2021. IoT notary: Attestable sensor data capture in IoT environments. *ACM Transactions on Internet of Things* 3, 1 (2021), 1–30.

[46] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An embeddable analytical database. In *International Conference on Management of Data*. 1981–1984.

[47] G Anthony Reina, Alexey Gruzdev, Patrick Foley, Olga Perepelkina, Mansi Sharma, Igor Davidyuk, Ilya Trushkin, Maksim Radionov, Aleksandr Mokrov, Dmitry Agapov, Jason Martin, Brandon Edwards, Micah J. Sheller, Sarthak Pati, Prakash Narayana Moorthy, Shih han Wang, Prashant Shah, and Spyridon Bakas. 2021. OpenFL: An open-source framework for Federated Learning. arXiv:2105.06413 [cs.LG]

[48] Mark Russinovich, Edward Ashton, Christine Avanessians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Cédric Fournet, Matthew Kerner, Sid Krishna, et al. 2019. CCF: A framework for building confidential verifiable replicated services. *Technical report, Microsoft Research and Microsoft Azure* (2019).

[49] 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 *IEEE Symposium on Security and Privacy*. IEEE, 38–54.

[50] AMD SEV-SNP. 2020. Strengthening VM isolation with integrity protection and more. *White Paper, January* (2020).

[51] Alex Shamis, Peter Pietzuch, Miguel Castro, Edward Ashton, Amaury Chamayou, Sylvan Clebsch, Antoine Delignat-Lavaud, Cedric Fournet, Matthew Kerner, Julien Maffre, et al. 2021. PAC: Practical Accountability for CCF. *arXiv preprint arXiv:2105.13116* (2021).

[52] Yuanyuan Sun, Sheng Wang, Huorong Li, and Feifei Li. 2021. Building enclave-native storage engines for practical encrypted databases. *Proceedings of the VLDB Endowment* 14, 6 (2021), 1019–1032.

[53] SUSE. 2022. AMD SEV Guide. https://documentation.suse.com/sles/15-SP2/html/SLES-amd-sev/art-amd-sev.html. Online; accessed 28 February 2022.

[54] Miklos Szeredi. 2010. FUSE: Filesystem in userspace. *http://fuse. sourceforge. net* (2010).

[55] Carol Tenopir, Suzie Allard, Kimberly Douglass, Arsev Umur Aydinoglu, Lei Wu, Eleanor Read, Maribeth Manoff, and Mike Frame. 2011. Data sharing by scientists: practices and perceptions. *PloS one* 6, 6 (2011), e21101.

[56] UCI. 2022. Adult Income Dataset. https://www.kaggle.com/wenruliu/adult-income-dataset. Online; accessed 1 March 2022.

[57] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. 2019. Conclave: secure multi-party computation on big data. In *14th EuroSys Conference*. 1–18.

[58] Frank Wang, James Mickens, Nickolai Zeldovich, and Vinod Vaikuntanathan. 2016. Sieve: Cryptographically enforced access control for user data in untrusted clouds. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 611–626.

[59] Kang Wei, Jun Li, Ming Ding, Chuan Ma, Howard H Yang, Farhad Farokhi, Shi Jin, Tony QS Quek, and H Vincent Poor. 2020. Federated learning with differential privacy: Algorithms and performance analysis. *IEEE Transactions on Information Forensics and Security* 15 (2020), 3454–3469.

[60] Alan F Westin. 1968. Privacy and freedom. *Washington and Lee Law Review* 25, 1 (1968), 166.

[61] Alex Wong. [n.d.]. COVID-Net. https://github.com/AlexSWong/COVID-Net Online; accessed 21 May 2022.

[62] Jiaan Zeng, Guangchen Ruan, Alexander Crowell, Atul Prakash, and Beth Plale. 2014. Cloud computing data capsules for non-consumptiveuse of texts. In *5th ACM workshop on Scientific Cloud Somputing*. 9–16.

[63] 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 *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 283–298.

[64] 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 *30th USENIX Security Symposium (USENIX Security 21)*. 2723–2740.

[65] Jinwei Zhu, Kun Cheng, Jiayang Liu, and Liang Guo. 2021. Full Encryption: An end to end encryption mechanism in GaussDB. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2811–2814.

[66] Alexander Ziller, Andrew Trask, Antonio Lopardo, Benjamin Szymkow, Bobby Wagner, Emma Bluemke, Jean-Mickael Nounahon, Jonathan Passerat-Palmbach, Kritika Prakash, Nick Rose, et al. 2021. Pysyft: A library for easy federated learning. In *Federated Learning Systems*. Springer, 111–139.