

Device-centric Federated Analytics At Ease

Li Zhang, Junji Qiu, Shangguang Wang, Mengwei Xu
Beijing University of Posts and Telecommunications
Beijing, China

ABSTRACT

Nowadays, high-volume and privacy-sensitive data are generated by mobile devices, which are better to be preserved on devices and queried on demand. However, data analysts still lack a uniform way to harness such distributed on-device data. In this paper, we propose a data querying system – Deck, that enables flexible device-centric federated analytics. The key idea of Deck is to bypass the app developers but allow the data analysts to directly submit their analytics code to run on devices, through a centralized query coordinator service. Deck provides a list of standard APIs to data analysts and handles most of the device-specific tasks underneath. Deck further incorporates two key techniques: (i) a hybrid permission checking mechanism and mandatory cross-device aggregation to ensure data privacy; (ii) a zero-knowledge statistical model that judiciously trades off query delay and query resource expenditure on devices. We fully implement Deck and plug it into 20 popular Android apps. An in-the-wild deployment on 1,642 volunteers shows that Deck significantly reduces the query delay by up to 30× compared to baselines. Our microbenchmarks also demonstrate that the standalone overhead of Deck is negligible.

1 INTRODUCTION

Data is perhaps the most valuable asset in our digital world nowadays. This is especially the case for data generated by mobile and IoT devices, which help reveal user behaviour and physical worlds changes [65, 78]. To analyze data generated by massive devices, the mainstream methodology is *cloud-centric* [65], where the devices stream the data to clouds so data users¹ can analyze them with unified frameworks like Spark [118] and Flink [53].

Data staying on devices, however, is an emerging paradigm in recent years. The reasons are twofold. First, with the ever-growing public concerns over data privacy and relevant regulations (e.g., GDPR [3] and CCPA [4]), companies can no longer collect data arbitrarily as they used to. Second, the data volume is explosively growing, outpacing the upgrading speed of network capacity, especially the wide-area network (estimated to be 4× and 1.5× from 2018 to 2022, respectively [5, 7]). For instance, a recent work [111] shows 99% videos captured by a campus’ IoT cameras will not be queried and building an analytics system that retrieves data on demand can reduce the network cost by two orders of magnitudes. Such private and cold data are better to be preserved on devices. Only a small, critical portion of them shall be permitted (by privacy or network) to be streamed to clouds.

¹In this paper, we term those who query mobile data as *data users*. A data user can be a human (e.g., data analyst) or a running program (e.g., a recommendation system that relies on realtime user preferences for online decision [44, 58, 77]). To be noted, *app developers*, who build the app logic and data querying system, are often different from data users, e.g., they may come from different departments of the same company, or even from different companies for the scenario of cross-entity data sharing [82].

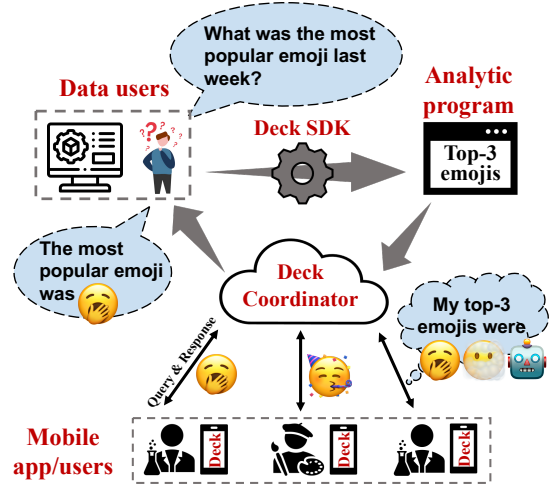


Figure 1: A use case of performing federated analytics.

The trend catalyzes *device-centric federated analytics* [6], as the data processing is shifted from clouds to devices and only privacy-friendly or more compact results are returned to clouds. Use cases abound: (1) Google, the early proposer of federated analytics, employs it in Pixel phones to identify the songs playing in a region with an on-device database previously distributed [40]. It enables Google engineers to improve the song database, e.g., 5% accuracy improvement globally as reported, without any phone revealing which songs were heard. (2) Mobile input methods typically record what user types on the keyboard on local storage for those who opt in [100]. As Figure 1 shows, at any time, a sociologist may issue a query to characterize the users’ input patterns, e.g., the most popular emoji used last week, to understand the society-scale reactions to certain public events. (3) Federated learning [50] is a special paradigm of device-centric federated analytics, where the devices collaboratively train a machine learning model without sharing the raw training data. Meanwhile, its testing process, i.e., to measure how an FL-trained model performs in the real world, is another important federated analytics scenario. For all above cases, keeping data on local storage not only maximizes privacy preservation, but also enables fine-grained per-user privacy configuration, e.g., each user could turn off their data accessibility at any time.

However, there lacks a flexible and uniform framework to perform device-centric federated analytics. The status quo is ad-hoc: (1) The app developers *hard-code* the analytics logic into the specific app to get post-analytics results to clouds; (2) A new query needs to undergo a *complete software engineering* process, i.e., app developers need to revise the app’s logic, run testing cases, and upgrade the app on devices, even though the query is quite simple. Such a process is labor-intensive, cumbersome, and inefficient. More advanced tools [8, 9, 21, 34] relieve the developers from rebuilding

the whole applications, but have not fundamentally addressed the issue of programmability in federated analytics.

As a novel concept, very few literature explores the field of federated analytics (FA) [105]. They mostly build prototype frameworks to demonstrate its feasibility, e.g., video analytics on network cameras [70], collaborative frequent pattern mining [108] or tackling data skewness in FA [107]. They are either designed for specific apps or solving dedicated type of problems, rather than providing a unified solution to perform FA in the wild. TensorFlow Federated [41] doesn't provide an end-to-end solution for FA on mobile devices. Earlier, there are closely related concepts to federated analytics, e.g., in-sensor-network processing [80, 116], wide-area-network analytics [86, 102], and distributed differential privacy analytics [89–91]. Those systems do not fit into our proposed scenarios for one or many of the following reasons: (1) They may ignore the privacy aspect of data query, which is a primary incentive to decentralize the data analytics. (2) They mostly target traditional SQL query optimizations, yet modern data analytics requires more flexible programmability, e.g., deep learning training or inference. (3) They either optimize for energy consumption (on sensors) or query speed (on edge servers),

To fill the gap, we propose Deck, the first-of-its-kind uniform framework for modern device-centric federated analytics. The key idea of Deck is to bypass the app developers, but allow data users to directly submit their code to run on devices. Deck provides a set of standard interfaces to data users mainly in Java, including Pandas-like DataFrame, ML-related operations, databases access, as well as Android-native function wrappers. Underneath, Deck handles most of the device-specific tasks transparently, thus allowing data users to focus on the data analytics logic as if the data has been already retrieved and locally stored. Deck is designed for *approximate data query* [42, 48, 55] that requires only a subset of devices to obtain a fairly confident result as shown in Figure 1.

Scope of this work As a new paradigm in data querying systems, there emerges manifold research questions to be solved. As the very early attempt, this work only probes into the sound architectural design of device-centric FA systems, and how it can possibly enhance the data analytics flow of existing apps. Technically, this work targets the fundamental demands in building a data querying system as traditional distributed/SQL systems [106]: (1) the querying performance, e.g., the query latency and resource cost. (2) the data privacy guarantee.

Deck consists of two core modules: (i) a central *Coordinator* that receives queries from data users, generates and schedules tasks to devices, retrieves and aggregates the results from the devices, and finally returns the aggregated result to data users; (ii) an *Execution Sandbox* that runs on devices and handles the tasks dispatched by *Coordinator*. To use Deck, the app developers only need to embed the *Execution Sandbox* into the app as an independent service, and deploys the *Coordinator* on cloud with a user bookkeeping system to manage how different data users can utilize the data, e.g., accessible datasets, maximal query frequency, etc.

Key techniques Deck's design in using a central coordinator for device dispatching and results collection also offers the opportunity to employ a consolidated policy to manage data usage. To design such a policy, however, we face two primary challenges.

- First, how to offer data users flexible programmability while guaranteeing data privacy? Since the analytics logic could be far more complicated than a SQL query as shown in Table 3, Deck is designed to allow almost arbitrary Java operations. In our threat model, *Coordinator* and *Execution Sandbox* are trusted, while data users can be curious and malicious, therefore various attacks can be potentially performed during a query, e.g., permission outreach through Java reflection.

- (§3) Deck employs multiple mechanisms to guarantee data privacy. To defend permission outreach, Deck combines (i) a lightweight *Annotation and Proxy* mechanism powered by Deck's defined APIs that all resource accessing operations will be redirected to; (ii) a hybrid code analysis technique to detect prohibited method usage in a query. Through above techniques, Deck protects sensitive data and APIs from unauthorized data users at both Java and native code. To defend differential analysis attacks, Deck enforces a query to use cross-device aggregation, so data users can access the statistical patterns of the queried dataset but not personal information.

- Second, how to trade off resource expenditure and query delay? An analytic query needs to involve enough number of devices (specified by data users in Deck) to be statistically meaningful. However, the query response time of every single device is volatile due to the dynamic network environment and device status as we experimentally demonstrate in §4.1.1. Therefore, sending a query only to the required number of devices often results in an unpredictable and long query delay. To mitigate such impacts from devices' dynamic status, it is straightforward to redundantly schedule a query to more devices than expected. Such redundancy, however, consumes precious hardware resources (e.g., network, compute and energy), therefore needs to be minimized.

- (§4) To better trade off the resource expenditure and query delay, Deck incorporates a statistical model to guide the device dispatching strategy in an *incremental manner*. The key idea is that, based on the progress of results being returned, we can calculate the expectation of the remaining time to finish a query. For delayed queries, Deck can opportunistically dispatch the analytics tasks to more devices to compensate for the delay from slow devices. The policy requires zero apriori knowledge about devices' hardware status and incurs no additional privacy or network cost.

Field deployment and evaluation We fully implemented Deck and embedded it into 20 popular open-source Android apps with diversified data queries. We conducted a field deployment by installing 3 Deck-enabled apps on 1,642 real users' mobile phones for 14 days. Through 3,517 queries and 232,779 responses from those devices, Deck shows superior performance: its 99th-percentile query delay is only 2 seconds on average, which is up to 30× smaller than using a fixed redundancy. Besides, developing with Deck is highly efficient, as one query requires only 77 lines of code on average across the 20 apps we built. Compared to industrial hot-fix libraries [34], Deck is up to 105× faster in dispatching a query for its decoupled query and app logic.

Contributions are summarized below.

- We propose a uniform framework, Deck, for device-centric federated analytics in a swift, flexible manner. Deck provides a list of easy-to-use programming interfaces to data users and hides the device-specific details.

- We address two key challenges to make Deck safe and efficient. One is through a comprehensive, hybrid security mechanism to ensure data privacy. The other one is through a statistical model that incrementally involves more devices for fast response.
- We implement Deck, plug it into popular Android apps, and deploy those apps on 1,642 users' devices to study their performance in the wild. The results demonstrate Deck's effectiveness in delivering fast query responses with little resource waste.

2 DECK DESIGN

2.1 Design principles and goals

When designing Deck, we make the following assumptions about data users.

- They have very little knowledge of mobile programming, but only standard data analytics languages, i.e., Java in our case to facilitate query execution on Android devices. Note that Java has extensive usage in big data processing frameworks like Spark [118] and Flink [53].
- They care little about the resource cost to run the query on devices. Somewhat paradoxically, they expect the query delay to be as short as possible, because a long delay annoys them, especially when the query is part of an online service [44, 58, 77].
- They have zero incentive to guarantee the privacy validity of their queries. In reality, the data users may be careless or even malicious, as the data might be shared among different parties to maximize their business returns.

Therefore, Deck needs to hide the device-specific details, trade off the resource costs and query delay, and impose privacy-preserving mechanisms in an automated and non-intrusive manner. It is achieved through its overall architectural design, API interfaces, and the key techniques presented in the following subsections.

Federated analytics is mostly *approximate* [42, 48, 55], meaning it does not require a complete dataset (from all devices) to acquire the final result; and Deck is designed for such approximate queries. Such queries are prevalent, as (1) iterating over each device can be exceptionally slow; (2) a small subset of data is often enough to generate a confident result [43, 60, 61]. According to our query to study the average time interval between users' typing sequences (Q1 in Table 3), we find that querying 30 users gives us a 10% confidence interval with a 95% confidence level.

2.2 Deploying Deck

Deploying Deck consists of two core stages.

- First, app developers need to plug Deck's device-side library into their apps and pair it with local datasets. We call them Deck-enabled apps. App developers may involve as many datasets that could be potentially used for analytics as possible, and Deck employs a user-dataset permission control system to allow a user to access only certain datasets with granted permission. A Deck-enabled app will maintain a persistent connection with the central *Coordinator*, which is usually deployed on a cloud server, for instant task/result message exchanging.
- Second, app developers deploy a central *Coordinator* on a server that handles queries from data users and interacts directly with devices for task dispatching. It works as an intermediary between data users and the devices. Such a centralized design opens huge

space for a unified design to trade off the query delay and on-device resource costs, as well as imposing the query privacy validation. The scalability of Deck, e.g., to more data users or devices, can be effectively expanded by scaling the hardware capacity of the servers where *Coordinators* are deployed.

2.3 APIs and use cases

Deck's programmability is based on Android-compatible Java features². It also means the data users can use any compatible third-party Java libraries. Besides, data users can also use JNI technology for programming with native C++ code, as used in our federated learning query to facilitate on-device model training.

In addition, Deck provides a set of simple programming interfaces (shown in Table 1) to facilitate data users to write data queries: (1) Deck's system-level APIs that facilitate data users to dispatch and manage the analytic tasks. An analytic task is specified with a given number of devices to run through *Deck.task*. The execution function *Task.run* is non-blocking until the *Task.get* is invoked. (2) Deck's application-level APIs that run on the *Coordinator*, e.g., *mean*, *sum*, *aggregateby* for *DataFrame* and *FL.aggModel* for *FL*, etc. We retrofit the design of *Pandas* [81] and *MNN* [71] to improve the usability of these APIs. (3) Deck's Android wrappers for processing different types of data such as databases, images, and media. Those APIs for Android platform are typically implemented more efficiently as compared to Java APIs if there exist. They also implement Deck's *Annotation and Proxy* mechanism, responsible for permission checking and avoiding out-of-boundary resource accessing by data users, who are prohibited to use the origin Android APIs directly. All above APIs are packed into a Java SDK for data users, with which data users only need to focus on implementing the data analytic logic.

```
class OnDeviceActor extends Actor {
    @DeckFile("datasetPath")
    public static Result<Model> run(Model model, int epoch) {
        Optimizer optimizer = new Optimizer("SGD");
        for (int i = 0; i < epoch; ++i) {
            model = trainIter(model, optimizer, "datasetPath");
            // Train model using MNN native method
        }
        return new Result<>(model);
    }
}

public class DeckSys {
    public static void main(String[] args) {
        Deck.init("ip:port", "credential", false);
        Model model = FL.loadModel("initModel");
        for (int i = 0; i < 40; ++i) { // Rounds
            Task task = Deck.task(DeviceTrain, 50);
            // Aggregate models from 50 devices
            task.run(model, 5); // Non-blocking invoke
            FL.ModelRef modelRef = FL.aggModel(task);
            // Get model reference immediately
            model = task.get(modelRef, 60*10)["model"];
            // Blocking to get aggregated model
        }
        FL.saveModel(model, "trainedModel");
    }
}
```

Listing 1: A federated learning query.

²Android supports all Java 7 and most Java 8 features.

Type	Interface	In	Out	Description
Java API	All Java 7 and a subset of Java 8 features/APIs are supported. Java-based third-party libraries are also supported, such as Gson and SLF4J.			
Deck System-level API	<i>Deck.init</i>	url (Str), credential (Str), debug (Bool)	None	Init Deck with coordinator url, user credential and debug option.
	<i>Deck.task</i>	taskClass (Class), targetNum (Int)	Task	Construct a new task with class containing on-device logic.
	<i>Deck.@Annotation</i>	resourcePath (Str)	None	Explicitly annotate resources that data user will use in a query. For example: <i>@DeckFile</i> , <i>@DeckDB</i> , <i>@DeckImage</i> , etc.
	<i>Task.run</i>	params (Object[])	None	Pass runtime parameters and dispatch task to Android devices.
	<i>Task.get</i>	ref (Task.ObjRef), timeout (Int)	Map<String, Object>	Block to get result according to object reference (e.g., DF.RetRef, FL.ModelRef) until timeout.
Aggregation	<i>DF.statsOP</i>	colName (Str)	DF.RetRef	Mathematical reduce operations on the column of DataFrame.
	<i>DF.aggregateby</i>	colName (Str)	DF.AgBy	Combine entries with the same value on the column and return an aggregated intermediate representation.
	<i>DF.AgBy.statsOP</i>	colName (Str)	DF.RetRef	Mathematical reduce operations on the column of DF.AgBy object.
FL	<i>FL.aggModel</i>	FLTask (Task)	FL.ModelRef	Aggregate multiple FL models and return a model reference.
	<i>FL.saveModel</i>	model (FL.Model), modelPath(Str)	None	Serialize model and save to target path.
	<i>FL.loadModel</i>	modelPath (Str)	FL.Model	Load model from local path.
Android Wrapper	<ol style="list-style-type: none"> [Android.database.sqlite] SQL-related operations: <i>getDB (Proxy method)</i>, <i>checkPermission</i>, <i>sqlQuery</i>, <i>close</i>, etc. [Android.graphics.Bitmap] Image-related operations: <i>getImage (Proxy method)</i>, <i>getWidth</i>, <i>getHeight</i>, <i>getPixel</i>, etc. [Android.media] Audio/video-related operations: <i>getMedia (Proxy method)</i>, <i>getMediaDuration</i>, <i>encodeVideo</i>, <i>encodeAudio</i>, etc. 			

Table 1: Deck’s primary APIs provided to data users.

A case of federated learning is shown in Listing 1. In the code snippet, Class *OnDeviceActor* contains the analytic logic to be run on devices, and Class *DeckSys* contains the management logic using Deck’s system-level APIs, which will be executed on data user’s desktop. This program first disables debug mode and connects to *Coordinator* with credential file using *Deck.init*. In each round, we construct a *Task* object using *Deck.task*. The task will be dispatched to devices by *Coordinator* once *task.run* is called and all local parameters will be transferred to *Coordinator*. We can use Deck’s FL API to invoke model aggregation across trained models returned from each device by passing a *Task* object as an identifier. Once getting an aggregated model using *task.get* from the *Coordinator*, we can test the accuracy or run other FL-related operations upon it locally before entering the next training round. At last, we save the trained model to our local desktop.

2.4 Workflow

Figure 2 shows how Deck works internally.

Local compiling A query begins with data users running the query on their local machine, e.g., a desktop. The device-side query code will be compiled to Java-Class files locally, then uploaded to *Coordinator* with data user’s credential, third-party dependencies, and the parameters. All the processes will be performed automatically by Deck’s Data-user SDK.

User bookkeeping Upon receiving a query request, the *Coordinator* first authenticates the data user and checks if there is still enough “quantum” left for her. The quantum is defined as how many devices have been queried by this data user in a past period, e.g., a month. The use of quantum is to prevent data users from excessively submitting queries, and potentially be used in a billing model (not the focus of this work).

Privacy pre-checking *Coordinator* then performs permission checking on the submitted query after compiling the submitted Java-Class files to dex files in two steps: static permission checking to find the prohibited API usages and violated data access operations;

dynamic code injection to monitor the runtime behavior on devices. The details are discussed in §3.

Task scheduling Deck begins to dispatch the revised query to devices for execution. These devices are selected from a device pool that *Coordinator* maintains for those available devices. In order to optimize resource expenditure and query delay, a task scheduling policy is needed to determine *how many* and *what* devices the task should be dispatched to. Deck now employs a statistical model as described in §4. Not all queries will be dispatched to real devices – Deck allows the query to run in *debug* mode specified in the *Deck.init* API, so the query will only run on *Coordinator* with dumb data. It facilitates data users to test their code efficiently.

On-device execution Deck’s execution sandbox runs a task in a low thread priority to avoid compromising user experience in using the mobile device. The task is executed till completion unless: (i) the permission inspector is notified of a runtime permission violation by the code injected at *Coordinator*; (ii) the device is notified by the *Coordinator* that the query is completed with enough results or canceled by the data user.

Results aggregation The FA results returned from devices will be transmitted to *Coordinator* and aggregated across devices. Such aggregation is performed continuously with incoming results in a non-blocking manner to reduce the aggregating latency. When the results meet the targeting number, it returns the aggregated results to data users. Such post-aggregation data is often much less privacy-sensitive and more compact than the raw data.

3 PRIVACY GUARDING

3.1 Threat model

In our threat model, the deployed Deck runtime (including both *Coordinator* and *Execution Sandbox*) is trusted, while the data users may be curious or malicious. Such a threat model is commonly employed in practice, because the developers of popular apps often represent big companies, and are legally obligated to preserve data privacy [3, 4]. Instead, the data users can be any third-party entities

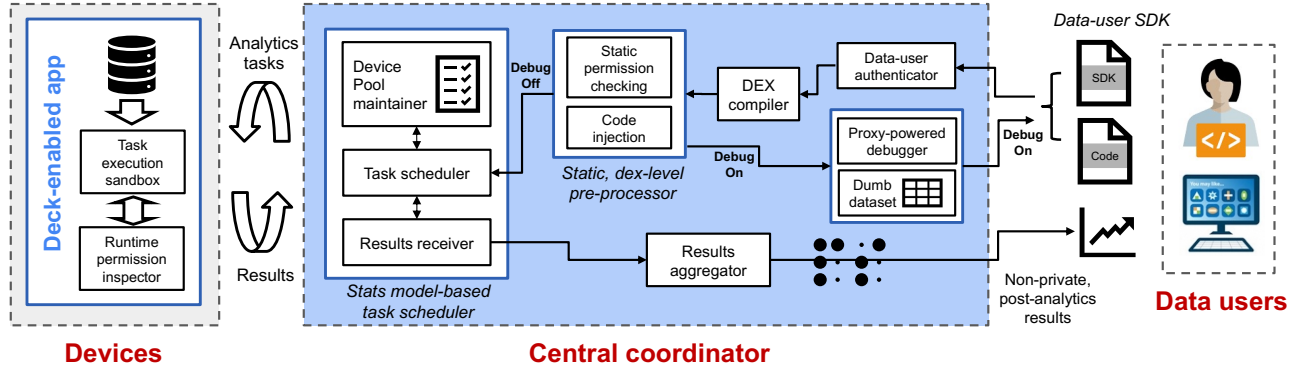


Figure 2: The workflow of Deck.

or individuals who access the data service through business cooperation or even payment [82]. The design of Deck can be extended to untrusted app developers as well, i.e., by delegating the *Coordinator* service to trustworthy governments.

It’s to be noted that the architectural design of Deck inherently includes certain privacy vantages in the FA scenario: (1) Data staying on devices allows device-owners to hold control of their accessibility, e.g., they can turn off the accessibility to their historical data; (2) Data users cannot specify which devices to execute their analytics, thus can not map the returned results to certain end users without external knowledge. (3) The *Coordinator* keeps track of each query code, thus the privacy violation can be verified afterward. Still, there is plenty of room for data users to obtain out-of-bound private information. In this work, we consider two main categories of attacks: permission outreach and differential analysis.

3.2 Defending permission outreach

While Deck allows data users to write almost arbitrary Java code, somehow the access is restricted for privacy preservation in two aspects: data (databases, images, etc.) and APIs (Android geolocation services, audio recording, etc.) Deck needs to guarantee that the unqualified queries not satisfying those policies will be rejected, and deal with potential intentional attacks like bypassing static code checking through Java reflection mechanism. The design matches Android’s permission system that aims to protect access to restricted data and actions [19].

3.2.1 Design choices. We first discuss existing techniques that can potentially defend the permission outreach.

In-app privilege isolation [87, 94] has been recently explored to provide a separated, controllable permission environment for dynamically loaded code from the original context. However, this approach requires modification to the Android framework, while Deck is an app-level framework to be paired with existing Android apps, thus making it undoable.

Proxy mechanism [69, 96] is widely used in software testing, especially for web applications [33, 98]. The key idea of proxy mechanism is to redirect privacy-sensitive methods to self-defined privacy-guaranteed methods. We can leverage Deck’s programming APIs to implement this conveniently.

Static code analysis [79] technique can detect the direct usage of Java classes and methods, but the malicious attackers can still

bypass it through Java reflection. With the most advanced approach, it is still challenging to restore the concrete inputs to a given function [74], i.e., the class/method name in our case. On the other hand, simply disabling the use of Java reflection leads to the unavailability of many third-party libraries (e.g., *kryo* reflects the "sum.misc.unsafe" class to speed up serialization [1]), which can severely compromise the programming flexibility of Deck.

Dynamic code analysis is a way to intercept concerned APIs and obtain the low-level runtime information, e.g., the classes/methods being reflected [45, 47, 49, 99]. A plausible solution is to deploy the intercepted systems on our own devices or emulators along with *Coordinator* for dynamic analysis. This approach, however, blocks the task scheduling because the tasks must be dispatched after the checking is completed, therefore introducing additional query delay. More than that, the code coverage of dynamic analysis is still limited and the malicious code might be bypassed [85].

3.2.2 Protecting data access.

Java code Deck employs a lightweight *Annotation-Proxy* mechanism [33, 98] powered by the design of Deck’s APIs (Table 1) to avoid malicious data accessing. More specifically, data users are required to explicitly annotate what data they will use (e.g., a dataset file or a folder of images) in their query, i.e., `@DeckFile` in Listing 1. These annotations will be extracted and checked by *Coordinator*. The queries will be rejected immediately if they request to access the data that the corresponding data users do not have permission to. Data users are also required to use Deck’s APIs to access local data, which is guaranteed by Deck’s technique described in the next subsection. These methods, acting as a *Proxy*, will check if the query is accessing the not-annotated data at query runtime on devices. If so, the query will also be rejected.

Native code Deck also allows data users to write or use third-party libraries with native methods, which are compiled into shared libraries. To avoid privacy leakage through native code, Deck leverages *isolatedProcess* [22], a unique Android feature that enables permission isolation of certain processes from the app. Deck puts the native code in an isolated process with zero permission, including network, read or write external storage, etc. Instead, the native code will be redirected to Deck’s proxy methods as discussed above through inter-process communication to access their granted data. Similarly, the proxy will check if data access is permitted.

3.2.3 Protecting API access.

Java code A query may access protected APIs through direct invoking or reflection. Deck detects direct invoking of the protected APIs through static code analysis [79] at *Coordinator*. For reflection, Deck uses dynamic code analysis [47, 49] to conduct dynamic permission inspection by ahead-of-time code injection. More specifically, Deck injects a line of code for class checking for every reflection usage, as shown in Listing 2. The injected code invokes an external function of Deck when running on devices, which looks up the class name in the blacklist. If matched, the device will abort the query immediately and send a violation code to *Coordinator*. Both the static code analysis and code injection run at dex level. They require no modification to Android framework or Linux kernel, making Deck compatible with any apps and third-party libraries. Deck disables dynamic library loading within the query, because it can bypass the code checking and injection of Deck.

```
String bad_class_name = "android.os.Environment";
Deck.runtime_checker.check(bad_class_name);
// <- Injected. The class will be checked to a blacklist.
File privateDir = (File) Class
    ..forName(bad_class_name)
    .getMethod("getDownloadCacheDirectory")
    .invoke(null);
```

Listing 2: A code injection example to be executed on Android devices, aims to detect disabled Class usage through reflection.

Native code In native methods, the sensitive operations may access any Java method through JNIEnv variable, or access the system-level method to open the file or access memory directly as we described above. We can redirect all sensitive operations to corresponding Java methods for permission checking. However, this requires extensive work for implementing the proxy methods for all sensitive operations in Deck’s API. As a result, we only consider the feasibility and implement some of the proxy methods. Deck also disables the method for loading a dynamic library using `dlopen` in native code.

3.3 Defending differential analysis

While the results returned to data users are post-analytics, they may still contain private information at a fine granularity not acceptable to be exposed to non-trusted parties. Such possibility of information leakage can be amplified in differential cryptanalysis attacks [73], where an attacker repeatedly issues a query and jointly analyzes the difference of returned results with external knowledge. For example, if an attacker can access the logs of specific websites with device identifiers, he can easily match the existing information with the query result to locate a particular device, even though the query result does not contain device-specific information.

To defend against such attacks, Deck employs a simple yet effective mechanism: *mandatory cross-device aggregation*. A valid query must end with the specified aggregation APIs, such as *sum*, *count*, or other aggregation operations. Otherwise, the query will be rejected. Therefore, the results returned to data users only reveal the statistical patterns of the queried dataset, while exposing no private

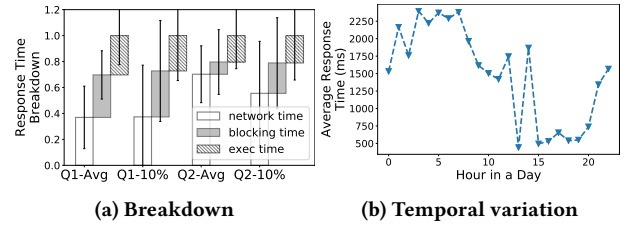


Figure 3: Response time analysis.

information of individuals. In federated learning, an aggregated model is commonly treated as non-private data [51] as opposed to the models trained on each device, which can be peeked into by an attacker through techniques like membership inference attack [95]. Deck also requires a query to target a pre-defined minimum number of devices (10 by default).

4 TASK SCHEDULING

Deck needs to minimize the query delay and resource expenditure – two critical metrics to any data querying system. In this section, we first show how this task is challenging, and then present Deck’s solution.

- **Query delay** is the end-to-end latency between data users submitting a FA query and receiving final results. It mainly includes three parts: network delay between data users and *Coordinator*, the pre-processing time at *Coordinator*, and the task scheduling time to dispatch tasks and waiting for enough results to be returned. However, the task scheduling time often dominates the overall query delay (shown in §6). Therefore, we mainly focus on optimizing this part of delay.

- **Resource redundancy** refers to how much on-device resources have been wasted during a query. Intuitively, the resource expenditure to run a query is multifold: network traffic, CPU cycles, battery consumption, etc. For simplicity, we don’t differentiate them but treat each device that ever runs the analytics task equally. Therefore, we define the resource redundancy as the extra percentage of devices that run the analytics task (either finished or not) to the targeted device number: $(actual_dev_num / target_dev_num) - 1$.

4.1 Query delay: a quick look

4.1.1 Preliminary Measurements. To understand the query delay, we first conduct a preliminary measurement on the query delay based on the data described in §6.1.

Breakdown The response time of each device is comprised of three parts: (1) the network delay, including task downloading and results uploading (*network time*); (2) the execution time of analytics tasks on devices (*exec time*); (3) the waiting time of the analytics tasks to be executed on devices (*blocking time*). Figure 3a shows that each part contributes a nontrivial portion to the end-to-end response time.

The long tail of response time A key observation is that the response time highly skews – the 99th-MAX response time is 37,167ms and 21.5× as high as its average. Such a long tail delay challenges the reliability of query performance. We find the reasons are multifold: (1) Devices with different data distribution and hardware capacity cause diverse on-device execution time. In our

federated learning query, this can cause an upmost gap of larger than 100× among devices. (2) There are high dynamics of network delay across time. As shown in Figure 3b, the average response time varies a lot within 24 hours of experiments, e.g., from 441ms to 2,397ms. Such dynamics are prevalent in nowadays Internet, due to the wireless signal change, device movement/handoff and wide-area-network congestion, etc [57]. (3) The device usage patterns cause the analytics tasks to be scheduled in a volatile way. Since the device-side tasks of Deck run in low priority to not interfere with user-interactive tasks, how much CPU time it can preempt depends on OS’s scheduling policies [39].

4.1.2 Design choices. Scheduling analytics tasks to devices to trade off query delay and resource expenditure is a brand-new research problem. Therefore, we first discuss several plausibly doable solutions yet are unemployed in Deck.

One-time dispatching with fixed redundancy An intuitive scheduling approach is to dispatch the task to more than needed to mitigate the impacts from long-tailing devices. For instance, Google’s federated learning system uses 30% fixed redundancy [50]. However, one-time dispatching with a fixed redundancy is far from optimal due to the high skewness of response time across devices and time as aforementioned: (1) With 20% fixed redundancy, the 99th-MAX query delay is 12,890ms, which is 14× higher than the median (920ms); (2) With 10% fixed redundancy, the query can’t even complete within our query timeout (100 seconds).

Discriminative task dispatching One may replace the random task dispatching with a discriminative one by predicting which devices are likely to respond fast. For example, the device hardware capacity, runtime status, and network conditions can be profiled periodically and used as the indicator of device response time. We omit such discriminative device dispatching policy for three reasons. First, collecting device runtime information to predict the response time incurs additional privacy and network costs, more or less, which is against the original intention of this work. Second, the response time of devices is volatile and depends on a variety of factors as aforementioned. Those factors are often out of Deck’s control and difficult to predict. Third, dispatching tasks to devices in a discriminative way may introduce statistical bias, resulting in unfaithful query results that are way off the ground truth.

4.2 Zero-knowledge statistical model

To tame the long tail of query delay, Deck is based on the idea that the query progress can be monitored on the fly, i.e., the percentage of returned result number as the query goes. According to this, we build a statistical model, which makes Deck opportunistically dispatch the task to more devices once it finds the progress delayed.

4.2.1 Skeleton of the model. At the beginning of a query, Deck dispatches the analytics task to the exact number of devices as needed without redundancy. As the query goes on and results keep coming, Deck periodically wakes up and dispatches the task to more devices (we call this wakeup interval a dispatching round). Based on the up-to-date feedback (the number of returned results) and the response time distribution (the probability density function), we build a statistical model to determine the number of additional devices to dispatch per round. The key idea of the statistical model

Symbol	Description
Z	Target device number specified by data users.
\mathcal{N}	Distribution of the historical response time of the query.
F	Cumulative density function (CDF) of distribution \mathcal{N} .
η	Solely tunable parameter to control the trade-off between query delay and resource expenditure.
t	Timestamp during task scheduling process.
k	Number of devices to be dispatched in a round.
$R(t)$	Returned results number at timestamp t .
$E(t)$	Expectation of returned results number at timestamp t .

Table 2: Symbols used in §4.2.

is to estimate *the reduction of query delay if dispatching the task to k new devices*. Deck chooses a proper k to ensure that it’s worth spending extra resource expenditure to speed up the query. There are several notable design choices made by Deck.

- Deck *incrementally* dispatches the analytics task to more devices instead of one-shot dispatching. This design can take advantage of the runtime feedbacks as the query results are continuously coming at various timestamps.
- Deck is *knowledge-free*, as it requires zero runtime information of devices, e.g., network conditions.
- Deck *randomly* selects devices to be dispatched, therefore no bias is introduced in the device selection stage.

4.2.2 Formulation. We propose our pseudo code in Algorithm 1 and used symbols in Table 2.

Algorithm 1: Deck’s task scheduling.

Initialization:

- Initial timestamp: $t = 0$
- Device number to be dispatched: $k = 0$
- Current results number at time t : $R(t)$
- Expectation of results number at time t : $E(t)$

Input:

- Deck’s wakeup interval: $intvl$
- Tuned threshold: η
- Target device number: Z

Output: Scheduling decisions during the task.

while $R(t) < Z$ **do**

$t_0 = \text{BinarySearch}(E(t_0) \approx Z)$

foreach k_i in $\{k_1, k_2, \dots, k_n\}$ **do**

$t_{k_i} = \text{BinarySearch}(E(t_{k_i}) \approx Z)$

$k = \max(k, k_i)$ if $\frac{t_0 - t_{k_i}}{k_i} > \eta$

end foreach

if $k > 0$ **then**

 Dispatch the analytic task to k devices

end if

$t = t + intvl$

end while

Assumptions and preliminaries Here we assume the response time of different devices is an *independent random variable* that follows certain distribution \mathcal{N} . Note that we do not presume \mathcal{N} to be any standard distribution like Gaussian distribution, but built it

from historical queries. Without any extra information, the expectation of a task to be returned after time slot t since dispatched can be estimated by the cumulative density function (CDF) of \mathcal{N} , termed as $F(t)$ [68]. However, at timestamp t' after t , the *Coordinator* is actually aware of whether the result has been returned during the time interval $[0, t']$. If not yet, the expectation of the result being returned at time interval $[t', t]$ can be calibrated as $\frac{F(t)-F(t')}{1-F(t')}$.

Modeling At the current dispatching round at timestamp t , *Coordinator* has got $R(t)$ results. If $R(t)$ is larger than the target device number Z , the query completes. Otherwise, Deck estimates how many devices it needs to send the task to. To make this decision, we first model the expected number of the returned results $E(t_{fut})$ in a future timestamp t_{fut} . Assuming that (1) we have dispatched the task to m devices, among which $r = m - R(t)$ devices have not yet returned the results; (2) the remaining r devices are dispatched at timestamp $\{t_1, t_2, \dots, t_r\}$ ($0 \leq t_i < t$). So $E(t_{fut})$ includes three parts: (1) the $R(t)$ results have been returned till time t ; (2) the results might be returned from the remaining r devices and (3) the results might be returned from the newly dispatched k devices at time t_{fut} . $E(t_{fut})$ can be formulated as follows:

$$E(t_{fut}) = R(t) + \sum_{i=1}^r \frac{F(t_{fut} - t_i) - F(t - t_i)}{1 - F(t - t_i)} + \sum_{i=1}^k F(t_{fut} - t) \quad (1)$$

Solving Apparently, $E(t_{fut})$ is a monotonic function with t_{fut} , and we can simply use a binary search to find the t_{fut} so that $E(t_{fut}) \approx Z$. As such, t_{fut} can be regarded as the expectation of the query delay. According to Eq 1, t_{fut} relates to the number of newly dispatched device number k . It's straightforward to prove that with larger k , the expected query delay t_{fut} can be reduced. When we dispatch the task to zero or k devices, we can estimate t_0 and t_k as task finished time

$$E(t_0) \approx Z; E(t_k) \approx Z \quad (2)$$

Here, we need choose the largest k so that

$$\frac{t_0 - t_k}{k} \geq \eta \quad (3)$$

By tuning η , we can trade off query delay and resource expenditure: lower η favors lower delay yet higher expenditure. For example, we record the expectation of query delay reduction and the number of devices to be dispatched in a real dispatching round in Figure 4. We use θ to represent how aggressive the diminished query acceleration can be when dispatching the task to one more device. When we dispatch the task to more devices, we can see the reduction of query delay caused by each device is getting smaller. So to avoid incurring high resource expenditure, it is not worth dispatching tasks to as many devices as possible. For each query, we can set a manually tuned parameter η to adjusted the trade-off between resource cost and query delay. In each dispatching round, we can dispatch the analytics task to optimal k devices where $\theta \geq \eta$, e.g., 17 in Figure 4. With this fine-grained constraint on dispatching device numbers, we can achieve the trade-off between query delay and resource expenditure.

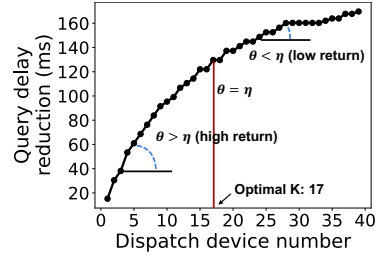


Figure 4: Diminished query acceleration with more dispatched devices.

5 IMPLEMENTATION

We have implemented a full prototype of Deck, which mainly comprises three modules.

- **Coordinator** is a standalone service deployed on a cloud server and uses Redis to save all metadata at runtime for the device connection pool, task scheduling, result retrieval, cached state, etc. Data users submit their query code to *Coordinator* in a compiled Java-Class file format with jar dependencies, which will be further compiled into separated dex files using *d8* command-line tool [10] for caching mechanism. The static code checking and code injection are implemented atop Soot [37]. In order to provide data users a uniform data format for future analysis, *Coordinator* constructs query results from devices to a Pandas-like DataFrame using joinery [54]. Later operations specified by data user's code will be performed on this DataFrame.

- **Android Runtime** mainly implements the device-side components of Deck, including the execution sandbox and runtime permission inspector. They are packed into an Android library for easy integration into existing Android projects. The communication between device and *Coordinator* is based on WebSocket protocol using OkHttp [32]. Persistent WebSocket connections are established between *Coordinator* and devices so Deck can maintain a device pool for task dispatching. When receiving a task, the device will deserialize it from the message and construct a new object to represent the task. We use Android WorkManager [39] to schedule new tasks to comply with OS's device-wise worker scheduling policy.

- **Data-user SDK** provides a list of Java interfaces for data users, containing interaction APIs with *Coordinator*, DataFrame-related operations, FL-related operations and a set of Android wrapper methods listed in Table 1. It provides data users easy-to-use, concise APIs, assuming they have limited knowledge of Android programming.

Optimizations We observed that the same or similar queries are often repeatedly issued, e.g., each round of training in federated learning. It motivates a caching mechanism to reduce the query cost. First, the *Coordinator* only runs the permission checker on the same dex file for one time. Second, each device allocates a fixed storage space (20MB by default) to cache the downloaded resources. It employs a least recently used policy for caching replacement. Only the dex not presented locally will be retrieved from *Coordinator*. We evaluate the effectiveness of caching mechanism in §6.

App category	App name	Query (before cross-device aggregation)	Instrumentation overhead		Runtime overhead (app in background)			Runtime overhead (app in foreground)		
			SLoC	APK Size (MB)	Energy (W)	CPU (%)	Network (KB/5Mins)	Energy (W)	CPU (%)	Network (KB/5Mins)
Keyboard	Trime	Average interval of typing sequences (Q1)	62	20.7→22.2	1.57→1.62	0→0	0→4.48	1.62→1.67	0→0	0→4.48
		Federated learning (Q4)	330	(↑1.5)	(↑0.05)	(0)	(↑4.48)	(↑0.05)	(0)	(↑4.48)
Email	K9 Mail	The number of attachment files in each inbox mail per day (Q2)	63	6.99→7.7	1.64→1.62	0→0.05	0→4.8	1.64→1.65	0→0	0→4.8
				(↑0.71)	(↓0.02)	(↑0.05)	(↑4.8)	(↑0.01)	(0)	(↑4.8)
Browser	Firefox	Average page loading time of certain url (Q3)	63	69.2→69.8	1.64→1.75	0.02→0.59	17.79→21.55	1.68→2.32	0.02→0.2	14.99→26.79
				(↑0.6)	(↑0.11)	(↑0.57)	(↑3.76)	(↑0.64)	(↑0.18)	(↑11.8)
Calendar	Simple Calendar	Application opening times per day	49	7.26→7.99	2.2→2.27	0→0	0→2.46	1.88→2.24	0→0.13	0→3.52
				(↑0.73)	(↑0.07)	(0)	(↑2.46)	(↑0.36)	(↑0.13)	(↑3.52)
Dialer	Simple Dialer	Dials-time distribution in a single day	58	5.68→6.35	2.06→2.1	0→0.03	0→4.07	2.19→2.23	3.06→3.08	0→4.07
				(↑0.67)	(↑0.04)	(↑0.03)	(↑4.07)	(↑0.04)	(↑0.02)	(↑4.07)
Messenger	Simple SMS Messenger	The average body length of SMS	51	7.85→8.64	2.31→2.37	0.02→0.03	0→4.17	2.28→2.34	0→0	0→4.12
				(↑0.79)	(↑0.06)	(↑0.01)	(↑4.17)	(↑0.06)	(0)	(↑4.12)
Photo Editor	Photo Editor	The average editing time of each picture	70	9.34→10.44	2.02→2.28	0→0.03	0→4.07	2.04→2.12	0→0	0→3.8
				(↑1.1)	(↑0.26)	(↑0.03)	(↑4.07)	(↑0.08)	(0)	(↑3.8)
Browser	DuckDuckGo	The number of websites in favorites	51	12.9→13.83	1.86→2.00	0.05→0.37	0→4.12	1.89→1.97	0.2→0.2	6.58→14.03
				(↑0.93)	(↑0.14)	(↑0.32)	(↑4.12)	(↑0.08)	(0)	(↑7.45)
Wiki	Wikipedia	The category of most visited K entries	65	11.5→12.33	1.58→1.55	0.05→0.23	0→5.57	1.67→1.64	0.25→0.29	2.89→7.24
				(↑0.83)	(↓0.03)	(↑0.18)	(↑5.57)	(↓0.03)	(↑0.04)	(↑4.35)
Game	DroidFish	Daily online time during a week	71	33.5→34.18	1.69→1.62	0→0	0→4.07	2.23→2.18	0.05→0.03	0→4.12
				(↑0.68)	(↓0.07)	(0)	(↑4.07)	(↓0.05)	(↓0.02)	(↑4.12)
Contacts	Simple Contacts	The number of contacts added last week	58	6.44→7.09	1.47→1.78	0→0	0→2.46	1.73→1.96	0.82→0.77	0→2.29
				(↑0.65)	(↑0.31)	(0)	(↑2.46)	(↑0.23)	(↓0.05)	(↑2.29)
Todo	ToDometer	The average completion time of todo entries	65	2.7→3.9	1.52→1.49	0.03→0.03	0→6.12	1.61→1.56	0.43→0.55	0→6.54
				(↑1.2)	(↓0.03)	(0)	(↑6.12)	(↓0.05)	(↑0.12)	(↑6.54)
Gallery	Simple Gallery	The average proportion of R/G/B pixels in each image drawn by user	145	14.7→16.19	1.58→1.54	0→0.02	0→2.46	1.76→1.73	0.24→0.3	0→2.46
				(↑1.49)	(↓0.04)	(↑0.02)	(↑2.46)	(↓0.03)	(↑0.06)	(↑2.46)
Clock	Simple Clock	The average number of repeated alarms before turned off by the user	71	6.46→7.16	1.59→1.56	0→0	0→2.46	1.64→1.64	0→0	0→2.46
				(↑0.7)	(↓0.03)	(0)	(↑2.46)	(0)	(0)	(↑2.46)
Music Player	Simple Music Player	The average music time and the corresponding favorite music categories	70	6.64→7.35	2.26→2.3	0→0	0→4.07	1.72→2.27	0.23→0.33	0→4.12
				(↑0.71)	(↑0.04)	(0)	(↑4.07)	(↑0.55)	(↑0.1)	(↑4.12)
Notes	Simple Notes	The frequency of creating a new note	51	6.08→6.69	2.16→2.28	0.02→0.02	0→3.96	2.11→2.24	0→0	0→4.12
				(↑0.61)	(↑0.12)	(0)	(↑3.96)	(↑0.13)	(0)	(↑4.12)
Reader	PDF Viewer Plus	The average reading time at morning/evening	59	20.3→21.77	1.58→1.54	0→0.05	0→4.8	1.63→1.58	0→0	0→4.8
				(↑1.47)	(↓0.04)	(↑0.05)	(↑4.8)	(↓0.05)	(0)	(↑4.8)
Sport	OpenTracks	The most frequently visited court	55	20.3→21.75	1.74→1.73	0→0	0→4.8	1.79→1.77	0→0.07	0→4.8
				(↑1.45)	(↓0.01)	(0)	(↑4.8)	(↓0.02)	(↑0.07)	(↑4.8)
Password Manager	KeyPass	The startup performance each time opening this app	63	4.94→6.14	2.41→2.42	0→0	0→3.47	2.42→2.41	0→0	0→2.46
				(↑1.2)	(↑0.01)	(0)	(↑3.47)	(↓0.01)	(0)	(↑2.46)
File Manager	Simple File Manager	The number of files deleted per day	50	6.21→6.91	2.3→3.39	0→0	0→2.46	2.32→2.32	8.9→12.45	0→2.46
				(↑0.7)	(↑1.09)	(0)	(↑2.46)	(0)	(↑3.55)	(↑2.46)
Average overhead			77	0.93	0.1	0.06	3.93	0.1	0.21	4.45

Table 3: The apps instrumented with Deck support and their overhead without query running under controlled experiments. Here we select 20 popular open-source apps [11–13, 15–18, 20, 23–31, 35, 36, 38] to demonstrate Deck’s good practicability.

6 EVALUATION

6.1 Experiment settings

Apps and queries As shown in Table 3, we plug Deck into 20 popular Android apps in diverse categories and design corresponding queries for each app.

Federated learning query We implement a federated learning query atop MNN [71], a lightweight DL library that supports on-device learning. We use the standard MNIST dataset and LeNet DNN as the learning task. According to our measurement, the training throughput of the devices involved in our field study varies from 110 to 1,040 frames/second. The dataset is split into Non-IID and downloaded when the input method is installed for one shot. The MNN library is dispatched to the user as part of Deck’s query, yet it’s cached during the training time according to Deck’s caching mechanism in §5.

Field deployment To investigate the usability and performance of Deck in the real world, especially the task scheduler, we recruited 1,642 volunteers to install three above Deck-enabled apps on their devices: input method, email client, and browser. The volunteers were recruited through a commercial third-party crowdsourcing

platform and compensated with a gift card. To ensure the responsiveness of queries, all volunteers were asked to use these apps by default. The *Coordinator* is deployed on a public server with 32 CPU cores, 64 GBs memory, and 200 MB network bandwidth capacity.

The field deployment lasted for two weeks in Jul. 2021. During the experiment, we used a long-running desktop program (as a data user) to periodically issue queries to devices for each app at an interval of 20 minutes. The first week was treated as the data collection stage, where we issue the query to all connected devices exhaustively. The second week was used for evaluating Deck’s task scheduling policy (§4). In total, we issued 3,517 queries and received 232,779 responses from devices. We observed only a small portion of devices can respond simultaneously, as the OS often goes to sleep when device is not in use. Among the 3 apps, the input method was the most responsive.

Parameter settings By default, we set the target device number to be queried as 100, which is often enough to have a statistically meaningful result, e.g., with a bounded error less than 5% at 95% confidence level for Q1. We set the wakeup interval in Deck’s scheduling algorithm as 100ms for SQL query and 1,000ms for FL query. We mostly focus on 10% and 20% of resource redundancy, slightly

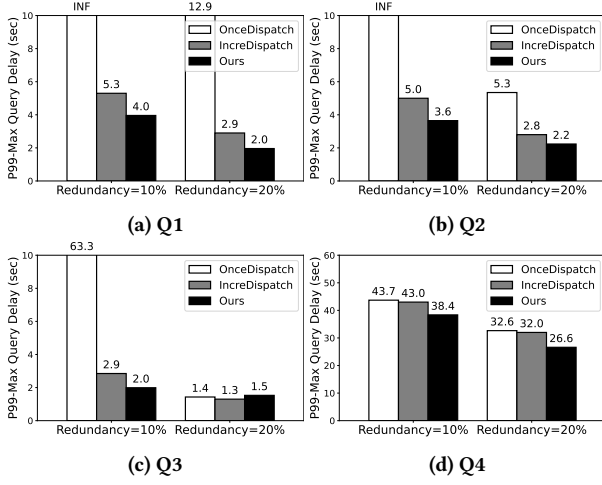


Figure 5: The end-to-end 99th-MAX query delay under various resource redundancy.

smaller than the configuration used in Google’s federated learning app [50]. We use a tighter resource budget because, in our vision, queries will be prevalent and frequently issued by many apps. Therefore, how we can push the limits in reducing resource redundancy is of primary importance.

Metric For query delay, we mostly report the value of 99th-percentile (99th-MAX) delay of all queries, as it’s a key metric widely adopted in data query systems [56].

Ethic considerations (1) This work is IRB-approved by the institution the authors are affiliated with. (2) We only collect post-analytics data from participants that are encrypted over TLS during transmission. (3) All data are anonymized during collection, storage, and processing.

6.2 End-to-end query delay

6.2.1 Breakdown. We first break down the end-to-end query delay of two queries (Q1 and Q2) using the fixed-redundancy dispatching algorithm. The results are summarized in Table 4. Cold/warm refers to whether the caching mechanism is invoked to avoid repeated pre-processing and downloading of third-party libraries. Overall, we find the task scheduling time dominates the query delay, contributing to more than 90% for both cold and warm queries. It motivates us to focus on reducing this time through a statistical model. In addition, our caching mechanism can reduce a large amount of pre-processing time on the *Coordinator*, for example, 322ms for Q1 and 386ms for Q2.

6.2.2 Task scheduling performance. We now evaluate our task scheduling policy presented in §4. We compare it with the two following baselines. • **OnceDispatch** simply dispatches task to devices with a fixed redundancy at the beginning of the query, which is used by traditional federated learning system [2].

• **IncreDispatch** borrows the idea from Deck to incrementally dispatch tasks according to the response from devices. The difference is that this baseline is not guided by a statistical model but the task scheduler will wake up and check how many results does it need to complete the analytics task each interval. Then a corresponding

Query type		End-to-end query delay breakdown (ms)		
		User-Coord network delay	Coord-side pre-processing	Task scheduling (target = 100)
Q1	Cold	104 (0.8%)	405 (3.0%)	12,964 (96.2%)
	Warm	29 (0.2%)	83 (0.6%)	12,890 (99.2%)
Q2	Cold	73 (1.2%)	476 (8.0%)	5,407 (90.8%)
	Warm	30 (0.6%)	90 (1.6%)	5,346 (97.8%)

Table 4: A breakdown of end-to-end query delay. “User”: data users; “Coord”: Coordinator.

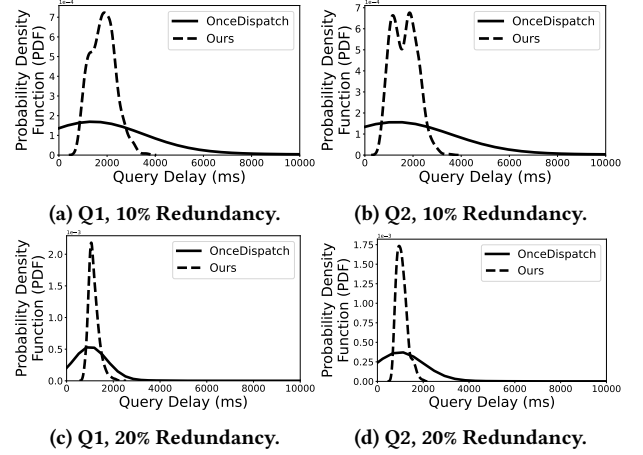


Figure 6: Probability density function (PDF) of query delay.

number of tasks will be incrementally dispatched to devices. We empirically choose the best parameters for this baseline.

Figure 5 illustrates the 99th-MAX query delay with two fixed redundancy of 10% and 20%. A key observation is that Deck significantly reduces the query delay as compared to the baselines. In Q1 and Q2, tasks can’t complete with 10% redundancy using OnceDispatch because the ultra-slow devices are dropped out in our experiments by a 100s timeout. Its degraded performance is due to the volatile, long-tail response time as observed in §4. In the most optimal case, Deck reduces the query delay by 31.65× (from 63.3s to 2s) in Q3. Deck also outperforms IncreDispatch by 1.12×–1.45× reduced delay for different queries and redundancy levels. The improvement mainly comes from Deck’s design of feedback-based and incremental task dispatching algorithm.

To further understand the overall distribution of the query delay, we show the probability density function (PDF) of two queries (i.e., Q1 and Q2) in Figure 6. As aforementioned, we select two redundancy level—10% and 20%, and compare our scheduling algorithm with OnceDispatch. The key observation is that our task scheduling algorithm can effectively mitigate the long-tail effects of query delay. Indeed, when we set the redundancy to 20%, our scheduling algorithm can achieve 2.45× and 3.43× lower 99th-MAX query delay for Q1 and Q2, respectively. The delay reduction becomes even higher with 10% redundancy, the 99th-MAX query delay of Q1 using OnceDispatch is 47,349ms and 15.15× slower than Deck.

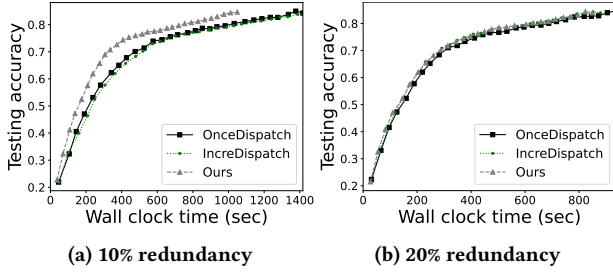


Figure 7: The federated learning performance.

6.3 A use case: federated learning

We also report the performance of federated learning, as an end-to-end use case for device-centric data analytics that involves multiple rounds of queries. We test the model testing accuracy and convergence time with 30-round training. The results are illustrated in Figure 7. With 10% resource redundancy, Deck is able to accelerate the convergence by $1.35\times/1.42\times$ as compared to OnceDispatch and IncreDispatch, respectively. The improvement is less profound with 20% resource redundancy, i.e., $1.10\times$ and $1.03\times$, respectively. Such results demonstrate the effectiveness of Deck’s task scheduling algorithm in the end-to-end, multi-round data queries.

6.4 System overhead

Standalone overhead We first measure the standalone runtime overhead when there is no query on the 20 Deck-enabled apps. The experiments are performed on a rooted Meizu 16T with Qualcomm Snapdragon 855 offline. For each app, we test the overhead for both cases when the app is running in foreground and background. For foreground testing, we turn on the device screen and stay on the app’s main page while for background we turn off the screen and wait for a minute before testing. Each testing lasts for 5 minutes and we report the total network traffic, the average CPU usage, and the average energy consumption. The network traffic is measured through `dumpsys netstats detail` command by filtering application uid. CPU usage is read through `top` command and energy consumption is read from `sysfs API`³ of Android platform. As summarized in Table 3, Deck incurs negligible standalone overhead: at most 1.5MB APK size and 1.09/0.64 watts of energy consumption in background/foreground. The CPU usage and network traffic increase an average of 0.06%/0.21% and 3.93KB/4.45KB in background/foreground, respectively. The overhead is primarily caused by the heartbeat messages from devices to *Coordinator*.

Query-time overhead We test the query-time overhead of Deck with the Deck-enabled input method, using the SQL query Q1 and the federated learning query Q4. We measured device’s energy consumption every 100ms and the start/end time of two queries. We use file size to be transferred through network of each query to represent network traffic. Results are illustrated in Figure 8. With the app in foreground, the real-time energy consumption can be increased by at most $5.05\times$ for FL query and $1.68\times$ for SQL query, as compared to standalone energy consumption. When the app is in background, the query-time overhead can be at most $8.09\times$

³We keep the device fully charged during experiments and read USB power supply from `/sys/class/power_supply/usb/`.

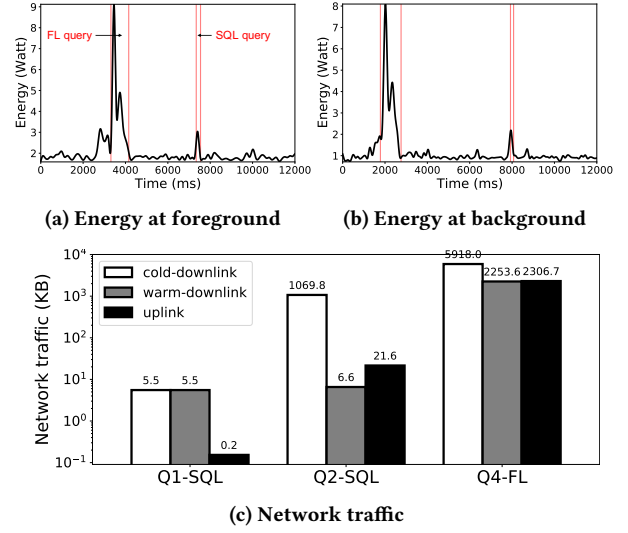


Figure 8: Query-time overhead of Deck.

and $2.61\times$. In addition, Deck incurs only a few KBs network traffic in SQL queries. An exception is that the downlink traffic of cold query in Q2 is around 1MB because a third-party library needs to be dispatched. The network usage of the federated learning query is as high as a few MBs to exchange the DNN model. Such nontrivial overhead motivates a task scheduling algorithm to minimize the devices being queried.

User survey Additionally, we conducted a survey on the participants involved in our field study. We asked whether they could feel the battery consumption increased during the 2-week field study. Among the 426 responses received, only 52 (14.6%) reported they could perceive the increased battery consumption. Note that during the experiments, we issued queries aggressively for data collection and system evaluation. In reality, a device is unlikely to receive too many queries in a short period of time.

6.5 Usability

We further study the usability of Deck as compared to a popular hot-fix library Tinker [34], which supports dex, library and resources update without reinstalling apk. To use Tinker, data users need to modify the code of app and generate the patch file to be dispatched to devices.

Table 5 summarizes the results. For queries using third-party libraries (e.g., image processing in our example), Deck generates a larger patch file, yet is much faster than Tinker to compile. This is because Tinker needs to compile the whole APK before generating the patch file, while Deck only needs to compile the query code. Overall, the end-to-end response time of Deck is much shorter than Tinker, i.e., $40\times$ for SQL query and $105\times$ for image processing query.

Additionally, Table 3 also summarizes the programming efforts paid by data users when using Deck. Overall, it only takes tens of lines of code to implement a query.

	Deck (SQL)	Tinker (SQL)	Deck (Image)	Tinker (Image)
Dispatch Size (KB)	2.53	5.04	407	151
Compile Time (s)	1.24	53.72	1.13	156.43
Network Time (s)	0.029	0.031	0.104	0.065
Exec Time (s)	0.125	1.951	0.272	2.219
Total (s)	1.394	55.702	1.506	158.714

Table 5: Comparison between Deck and Tinker with SQL and image processing queries.

7 RELATED WORK

Federated analytics As an emerging privacy-preserving data analytics paradigm, federated analytics has drawn attention from both industry [2, 40, 41, 100] and academia [70, 107, 108]. Unlike Deck that is designed for building generalized and arbitrary FA logic, previous work aim to (1) improve the user experience for specific mobile applications (e.g., predicting the next typed words in Gboard [2] or identifying the music playing around the user on Pixel phones [40]) (2) solve dedicated analytic problems, such as how to deal with data skewness in FA environments [107], or how to answer frequent pattern mining problems [108]. However, the primary target of Deck is to provide a uniform framework to simplify the process of designing, coding, deploying, and collecting results in FA analytics for mobile devices. Deck also uses lightweight but efficient mechanisms described in §3 to protect data privacy in our FA scenario. Additionally, the in-the-wild deployment shows the superior performance of Deck while others are mainly evaluated through controlled experiments.

Data-query permission checking Many efforts have been made to automatically enforce privacy policies on data queries, including approaches like tracking or restricting information flows in programs [52, 62, 84] and proposing new programming languages [64, 92, 115]. They are either tailored for specific query types (thus sacrifices programming flexibility) [93] or require instrumentation into the system runtime (undoable for Android OS) [114]. Instead, Deck employs a hybrid method (§3) to guarantee data privacy in best efforts. PrivacyStreams [76] protects users’ privacy by providing a list of APIs for app developers to access personal data, yet only targeting single device. Deck can leverage this work to diversify the on-device execution logic. In the future, Deck will retrofit those techniques to further enhance privacy preservation.

Distributed storage systems Partitioning data across multiple physical servers (nodes) has been a trend in recent system research due to the explosion of data volume [63, 66, 117]. The recent research work on revealing the burst edge sites also facilitates this paradigm [110]. For example, Ownership [106] automatically handles data movement for fine-grained task scheduling. DESEARCH [75] is a decentralized search engine that guarantees the integrity and privacy of search results for decentralized services. However, similar to WAN analytics systems [72, 86, 88, 102–104, 119], they focus on data placement and movement across servers instead of mobile devices, therefore ignoring the constrained hardware resource and the potentially malicious data queries.

Hot-fix libraries Android hot-fix libraries are mainly used to fix application bugs immediately without reinstallation or rebooting [8, 9, 21, 34] such as Tinker [34]. Compared with them, Deck provides unified and flexible programming interfaces directly to data users instead of app developers, and Deck is much faster in task compiling as it decouples the analytics logic from app logic as experimentally shown in §6.5.

Crowdsourcing platforms [67, 101, 113] enable developers or researchers to obtain data from a relatively large, geo-distributed group of participants. For example, Funf [14] is an open sensing framework that provides reusable functionalities to app developers for data collection, uploading, and configuration. Those frameworks are mostly built for data collection instead of querying systems, thus do not address the performance and privacy challenges as Deck does.

8 DISCUSSION

Privacy guarantee Powered by our self-designed APIs listed in Table 1, Deck currently aims to allow data users to reveal the statistical patterns of the distributed databases on clients. At the coordinator side, the privacy design forces data users to perform mandatory aggregation operation, which helps reduce the privacy leakage but also narrows the supported range of query types. To balance the privacy and programming flexibility, we can loose the mandatory aggregation constraint at the coordinator side, but enable local differential privacy [59] on each devices. Such a design also makes data users hard to get the concrete individual device data, but possible to operate on pre-aggregation data with a little data utility cost incurred by local differential privacy.

Deck can be potentially integrated with other advanced privacy and security techniques. For example, we can leverage the hardware-based solutions (e.g., ARM TrustZone and Intel SGX) to avoid data leakage from both device-side and server-side [83]. Isolating function execution environments is commonly used in cloud to power serverless computing by re-purposing the existing container platforms. However, container-based virtualization techniques [46, 97] fail in their coarse-grained isolating level (e.g., the overall Android filesystem), and thus are unable to apply to Deck. Recent work on isolating individual functions [109] may be a promising solution.

Usability enhancement Deck asks data users to specify the number of devices to be queried. In reality, it could be challenging for data users to give a “proper” device number that exhibits satisfactory accuracy and also low resource expenditure. We plan to extend Deck to accept the parameter as a confidence interval (e.g., 10%) and confidence level (e.g., 95%). For example, a data user may want to know the range (X, X+y) of the number of a song being played within a week. Deck will keep querying devices till the specified confidence is achieved. Confidence interval is extensively used in online query refinement [43, 112] and can further enhance the usability of Deck. Enabling such query mode requires Deck to revise its central *Coordinator* and the device scheduling algorithm.

9 CONCLUSIONS

In this work, we propose a unified framework Deck for elastic and efficient on-device federated analytics. Deck facilitates data users to focus on the analytics logic while handles the device-specific tasks

underneath. Deck also incorporates novel techniques to guard data privacy and trades off query delay with analytics resource expenditure on devices. The effectiveness of Deck is demonstrated through both large-scale field deployment and offline microbenchmarks.

REFERENCES

- [1] Java magic. part 4: sun.misc.unsafe. <http://mishadoff.com/blog/java-magic-part-4-sun-dot-misc-dot-unsafe/>, 2013.
- [2] Federated learning: Collaborative machine learning without centralized training data. <https://ai.googleblog.com/2017/04/federated-learning-collaborative.html>, 2017.
- [3] 2018 reform of eu data protection rules. https://ec.europa.eu/commission/sites/beta-political/files/data-protection-factsheet-changes_en.pdf, 2018.
- [4] California consumer privacy act (ccpa). <https://www.oag.ca.gov/privacy/ccpa>, 2018.
- [5] Cisco annual internet report (2018–2023) white paper. <https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html>, 2020.
- [6] Federated analytics: Collaborative data science without data collection. <https://ai.googleblog.com/2020/05/federated-analytics-collaborative-data.html>, 2020.
- [7] Global mobile data traffic from 2017 to 2022. <https://www.statista.com/statistics/271405/global-mobile-data-traffic-forecast/>, 2020.
- [8] alibaba/dexposed. <https://github.com/alibaba/dexposed>, 2021.
- [9] AndFix. <https://github.com/alibaba/AndFix>, 2021.
- [10] d8 | Android Developers. <https://developer.android.com/studio/command-line/d8>, 2021.
- [11] Droidfish. <https://github.com/peterosterlund2/droidfish>, 2021.
- [12] Duckduckgo android. <https://github.com/duckduckgo/Android>, 2021.
- [13] Firefox for android. <https://github.com/mozilla-mobile/fenix>, 2021.
- [14] Funf. <https://www.funf.org/about.html>, 2021.
- [15] K-9 mail. <https://github.com/k9mail/k-9>, 2021.
- [16] Keypass : Offline password manager. <https://github.com/yogeshpalaiyal/KeyPass>, 2021.
- [17] OpenTracks: a sport tracker. <https://github.com/OpenTracksApp/OpenTracks>, 2021.
- [18] Pdf viewer plus. <https://github.com/JavaCafe01/PdfViewer>, 2021.
- [19] Permissions on android. <https://developer.android.com/guide/topics/permissions/overview>, 2021.
- [20] Photoeditor. <https://github.com/burhanrashid52/PhotoEditor>, 2021.
- [21] Robust. <https://github.com/Meituan-Dianping/Robust>, 2021.
- [22] <service> | Android Developers. <https://developer.android.com/guide/topics/manifest/service-element>, 2021.
- [23] Simple calendar. <https://github.com/SimpleMobileTools/Simple-Calendar>, 2021.
- [24] Simple clock. <https://github.com/SimpleMobileTools/Simple-Clock>, 2021.
- [25] Simple contacts. <https://github.com/SimpleMobileTools/Simple-Contacts>, 2021.
- [26] Simple dialer. <https://github.com/SimpleMobileTools/Simple-Dialer>, 2021.
- [27] Simple file manager. <https://github.com/SimpleMobileTools/Simple-File-Manager>, 2021.
- [28] Simple gallery. <https://github.com/SimpleMobileTools/Simple-Gallery>, 2021.
- [29] Simple music player. <https://github.com/SimpleMobileTools/Simple-Music-Player>, 2021.
- [30] Simple notes. <https://github.com/SimpleMobileTools/Simple-Notes>, 2021.
- [31] Simple sms messenger. <https://github.com/SimpleMobileTools/Simple-SMS-Messenger>, 2021.
- [32] square/okhttp. <https://github.com/square/okhttp>, 2021.
- [33] stretchr/testify. <https://github.com/stretchr/testify>, 2021.
- [34] Tencent/tinker. <https://github.com/Tencent/tinker>, 2021.
- [35] Tedometer. <https://github.com/serbelga/ToDometer>, 2021.
- [36] Trime: Rime ime for android. <https://github.com/osfans/trime>, 2021.
- [37] Using Soot? Let us know about it! <https://github.com/soot-oss/soot>, August 2021.
- [38] Wikipedia android app. <https://github.com/wikimedia/apps-android-wikipedia>, 2021.
- [39] WorkManager | Android Developers. <https://developer.android.com/reference/androidx/work/WorkManager>, 2021.
- [40] Find out what music is playing near you | pixel phone help. <https://support.google.com/pixelphone/answer/7535326?hl=en>, 2022.
- [41] Tensorflow federated. <https://www.tensorflow.org/federated>, 2022.
- [42] Swarup Acharya, Phillip B Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. Join synopses for approximate query answering. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 275–286, 1999.
- [43] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42, 2013.
- [44] Marie Al-Ghossein, Tael Abdessalem, and Anthony Barré. Dynamic local models for online recommendation. In *Companion of The Web Conference 2018 on The Web Conference 2018*, WWW 2018, Lyon, France, April 23–27, 2018, pages 1419–1423. ACM, 2018.
- [45] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Stăicu. A survey of dynamic analysis and test generation for javascript. *ACM Computing Surveys (CSUR)*, 50(5):1–36, 2017.
- [46] Jeremy Andrus, Christoffer Dall, Alexander Van’t Hof, Oren Laadan, and Jason Nieh. Cells: a virtual mobile smartphone architecture. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 173–187, 2011.
- [47] Cyrille Artho, Viktor Schuppan, Armin Biere, Pascal Eugster, Marcel Baur, and Boris Zweimüller. Jnuke: Efficient dynamic analysis for java. In *International Conference on Computer Aided Verification*, pages 462–465. Springer, 2004.
- [48] Brian Babcock, Surajit Chaudhuri, and Gautam Das. Dynamic sample selection for approximate query processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 539–550, 2003.
- [49] Thomas Ball. The concept of dynamic analysis. In *Software Engineering—ESEC/FSE’99*, pages 216–234. Springer, 1999.
- [50] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingeman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečný, Stefano Mazzocchi, H Brendan McMahan, et al. Towards federated learning at scale: System design. *arXiv preprint arXiv:1902.01046*, 2019.
- [51] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1175–1191, 2017.
- [52] Niklas Broberg and David Sands. Paralocks: role-based information flow control and beyond. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages*, pages 431–444, 2010.
- [53] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.
- [54] Bryan Cardillo. joinery. <https://github.com/cardillo/joinery>, August 2021.
- [55] Kaushik Chakrabarti, Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Approximate query processing using wavelets. *The VLDB Journal*, 10(2):199–223, 2001.
- [56] Xusheng Chen, Haoze Song, Jianyu Jiang, Chaoyi Ruan, Cheng Li, Sen Wang, Gong Zhang, Reynold Cheng, and Heming Cui. Achieving low tail-latency and high scalability for serializable transactions in edge computing. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 210–227, 2021.
- [57] Karen Church and Nuria Oliver. Understanding mobile web and mobile search use in today’s dynamic mobile landscape. In *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services, MobileHCI ’11*, page 67–76, New York, NY, USA, 2011. Association for Computing Machinery.
- [58] M. Benjamin Dias, Dominique Locher, Ming Li, Wael El-Deredey, and Paulo J. G. Lisboa. The value of personalised recommender systems to e-business: a case study. In *Proceedings of the 2008 ACM Conference on Recommender Systems, RecSys 2008, Lausanne, Switzerland, October 23–25, 2008*, pages 291–294. ACM, 2008.
- [59] Úlfar Erlingsson, Vasyli Pihur, and Aleksandra Korolova. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 1054–1067, 2014.
- [60] Wenfei Fan, Floris Geerts, Yang Cao, Ting Deng, and Ping Lu. Querying big data by accessing small data. In *Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 173–184, 2015.
- [61] Wenfei Fan, Xin Wang, and Yinghui Wu. Querying big graphs within bounded resources. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 301–312, 2014.
- [62] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C Myers, and G Edward Suh. Verification of a practical hardware security architecture through static information flow analysis. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 555–568, 2017.
- [63] Daniel Ford, François Labelle, Florentina Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. 2010.
- [64] Daniel B Giffin, Amit Levy, Deian Stefan, David Terei, David Mazieres, John C Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *10th {USENIX} Symposium on Operating Systems Design and*

- Implementation ({OSDI} 12), pages 47–60, 2012.
- [65] Jiawei Han, Hong Cheng, Dong Xin, and Xifeng Yan. Frequent pattern mining: current status and future directions. *Data mining and knowledge discovery*, 15(1):55–86, 2007.
- [66] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage systems are distributed systems (so verify them that way!). In 14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20), pages 99–115, 2020.
- [67] Matthias Hirth, Tobias Hoffeld, and Phuoc Tran-Gia. Anatomy of a crowdsourcing platform—using the example of microworkers. com. In 2011 Fifth international conference on innovative mobile and internet services in ubiquitous computing, pages 322–329. IEEE, 2011.
- [68] Robert V Hogg, Joseph McKean, and Allen T Craig. *Introduction to mathematical statistics*. Pearson Education, 2005.
- [69] Jason I Hong and James A Landay. Webquilt: a framework for capturing and visualizing the web experience. In *Proceedings of the 10th international conference on World Wide Web*, pages 717–724, 2001.
- [70] Chuang Hu, Rui Lu, and Dan Wang. Feva: A federated video analytics architecture for networked smart cameras. *IEEE Network*, 35(6):163–170, 2021.
- [71] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, et al. Mnn: A universal and efficient inference engine. *arXiv preprint arXiv:2002.12418*, 2020.
- [72] Fan Lai, Jie You, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. Sol: Fast distributed computation over slow networks. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), pages 273–288, Santa Clara, CA, February 2020. USENIX Association.
- [73] Xuejia Lai, James L Massey, and Sean Murphy. Markov ciphers and differential cryptanalysis. In *Workshop on the Theory and Application of Cryptographic Techniques*, pages 17–38. Springer, 1991.
- [74] Davy Landman, Alexander Serebrenik, and Jurgen J Vinju. Challenges for static analysis of java reflection—literature review and empirical study. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pages 507–518. IEEE, 2017.
- [75] Mingyu Li, Jinhao Zhu, Tianxu Zhang, Cheng Tan, Yubin Xia, Sebastian Angel, and Haibo Chen. Bringing decentralized search to decentralized services. In 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21), pages 331–347. USENIX Association, July 2021.
- [76] Yuanchun Li, Fanglin Chen, Toby Jia-Jun Li, Yao Guo, Gang Huang, Matthew Fredrikson, Yuvraj Agarwal, and Jason I Hong. Privacystreams: Enabling transparency in personal data processing for mobile apps. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 1(3):1–26, 2017.
- [77] Hao Liu, Yongxin Tong, Panpan Zhang, Xinjiang Lu, Jianguo Duan, and Hui Xiong. Hydra: A personalized and context-aware multi-modal transportation recommendation system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2314–2324, 2019.
- [78] Mengchi Liu and Junfeng Qu. Mining high utility itemsets without candidate generation. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 55–64, 2012.
- [79] V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX security symposium*, volume 14, pages 18–18, 2005.
- [80] Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)*, 30(1):122–173, 2005.
- [81] Wes McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.
- [82] Microsoft. Dynamics 365: Cross-company data sharing. <https://docs.microsoft.com/en-us/dynamics365/fin-ops-core/dev-itpro/sysadmin/cross-company-data-sharing>, 2021.
- [83] Fan Mo, Hamed Haddadi, Kleomenis Katevas, Eduard Marin, Diego Perino, and Nicolas Kourtellis. Ppfl: privacy-preserving federated learning with trusted execution environments. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, pages 94–108, 2021.
- [84] Andrew C Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):410–442, 2000.
- [85] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. Dynamic malware analysis in the modern era—a state of the art survey. *ACM Computing Surveys (CSUR)*, 52(5):1–48, 2019.
- [86] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. Low latency geo-distributed data analytics. *ACM SIGCOMM Computer Communication Review*, 45(4):421–434, 2015.
- [87] Zhengyang Qu, Shahid Alam, Yan Chen, Xiaoyong Zhou, Wangjun Hong, and Ryan Riley. Dyddroid: Measuring dynamic code loading and its security implications in android applications. In 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pages 415–426. IEEE, 2017.
- [88] Ariel Rabkin, Matvey Arye, Siddhartha Sen, Vivek S Pai, and Michael J Freedman. Aggregation and degradation in jetstream: Streaming analytics in the wide area. In 11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14), pages 275–288, 2014.
- [89] Edo Roth, Karan Newatia, Yiping Ma, Ke Zhong, Sebastian Angel, and Andreas Haeberlen. Mycelium: Large-scale distributed graph queries with differential privacy. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, pages 327–343, 2021.
- [90] Edo Roth, Daniel Noble, Brett Hemenway Falk, and Andreas Haeberlen. Honeycrisp: large-scale differentially private aggregation without a trusted core. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 196–210, 2019.
- [91] Edo Roth, Hengchu Zhang, Andreas Haeberlen, and Benjamin C Pierce. Orchard: Differentially private analytics at scale. In 14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20), pages 1065–1081, 2020.
- [92] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.
- [93] Shayak Sen, Saikat Guha, Anupam Datta, Sriram K Rajamani, Janice Tsai, and Jeannette M Wing. Bootstrapping privacy compliance in big data systems. In 2014 IEEE Symposium on Security and Privacy, pages 327–342. IEEE, 2014.
- [94] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. Flexdroid: Enforcing in-app privilege separation in android. In NDSS, 2016.
- [95] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In 2017 IEEE Symposium on Security and Privacy (SP), pages 3–18. IEEE, 2017.
- [96] Sheikh Mohammed Sohan, Craig Anslow, and Frank Maurer. Spyrest: Automated restful api documentation using an http proxy server (n). In 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 271–276. IEEE, 2015.
- [97] Wenna Song, Jiang Ming, Lin Jiang, Yi Xiang, Xuanchen Pan, Jianming Fu, and Guojun Peng. Towards transparent and stealthy android os sandboxing via customizable container-based virtualization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2858–2874, 2021.
- [98] spring.io. Testing the web layer. <https://spring.io/guides/gs/testing-web/>, 2021.
- [99] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. Efficient dynamic analysis for node. js. In *Proceedings of the 27th International Conference on Compiler Construction*, pages 196–206, 2018.
- [100] Differential Privacy Team. Learning with privacy at scale. *Apple Machine Learning Journal*, 1(8), 2017.
- [101] Chris Van Pelt and Alex Sorokin. Designing a scalable crowdsourcing platform. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 765–766, 2012.
- [102] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. {CLARINET}: Wan-aware optimization for analytics queries. In 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), pages 435–450, 2016.
- [103] Ashish Vulimiri, Carlo Curino, P Brighten Godfrey, Thomas Jungblut, Jitu Padhye, and George Varghese. Global analytics in the face of bandwidth and regulatory constraints. In 12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15), pages 323–336, 2015.
- [104] Ashish Vulimiri, Carlo Curino, Philip Brighten Godfrey, Thomas Jungblut, Konstantinos Karanasos, Jitendra Padhye, and George Varghese. Wanalytics: Geo-distributed analytics for a data intensive world. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1087–1092, 2015.
- [105] Dan Wang, Siping Shi, Yifei Zhu, and Zhu Han. Federated analytics: Opportunities and challenges. *IEEE Network*, 36(1):151–158, 2022.
- [106] Stephanie Wang, Eric Liang, Edward Oakes, Benjamin Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. Ownership: A distributed futures system for fine-grained tasks. In NSDI, pages 671–686, 2021.
- [107] Zibo Wang, Yifei Zhu, Dan Wang, and Zhu Han. Fedacs: Federated skewness analytics in heterogeneous decentralized data environments. In 2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS), pages 1–10. IEEE, 2021.
- [108] Zibo Wang, Yifei Zhu, Dan Wang, and Zhu Han. Fedfpm: A unified federated analytics framework for collaborative frequent pattern mining. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 2022.
- [109] Nicholas C Wanninger, Joshua J Bowden, Kirtankumar Shetty, Ayush Garg, and Kyle C Hale. Isolating functions at the hardware limit with virtines. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 644–662, 2022.
- [110] Mengwei Xu, Zhe Fu, Xiao Ma, Li Zhang, Yanan Li, Feng Qian, Shangguang Wang, Ke Li, Jingyu Yang, and Xuanzhe Liu. From cloud to edge: a first look at public edge platforms. In *Proceedings of the 21st ACM Internet Measurement*

- Conference, pages 37–53, 2021.
- [111] Mengwei Xu, Tiantu Xu, Yunxin Liu, and Felix Xiaozhu Lin. Video analytics with zero-streaming cameras. In 2021 {USENIX} Annual Technical Conference ({USENIX}{ATC} 21), pages 459–472, 2021.
 - [112] Mengwei Xu, Xiwen Zhang, Yunxin Liu, Gang Huang, Xuanzhe Liu, and Felix Xiaozhu Lin. Approximate query service on autonomous iot cameras. In Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services, pages 191–205, 2020.
 - [113] Tingxin Yan, Matt Marzilli, Ryan Holmes, Deepak Ganesan, and Mark Corner. mcrowd: a platform for mobile crowdsourcing. In Proceedings of the 7th ACM conference on embedded networked sensor systems, pages 347–348, 2009.
 - [114] Chengxu Yang, Yuanchun Li, Mengwei Xu, Zhenpeng Chen, Yunxin Liu, Gang Huang, and Xuanzhe. Liu. Taintstream: Fine-grained taint tracking for big data platforms through dynamic code translation. In ESEC/FSE, pages 0–0, 2021.
 - [115] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In John Field and Michael Hicks, editors, Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012, pages 85–96. ACM, 2012.
 - [116] Yong Yao, Johannes Gehrke, et al. Query processing in sensor networks. In Cidr, pages 233–244, 2003.
 - [117] Jie You, Jingfeng Wu, Xin Jin, and Mosharaf Chowdhury. Ship compute or ship data? why not both? In 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21), pages 633–651. USENIX Association, April 2021.
 - [118] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12), pages 15–28, 2012.
 - [119] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A Lee. Awstream: Adaptive wide-area streaming analytics. In Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, pages 236–252, 2018.