OPAL: High performance platform for large-scale privacy-preserving location data analytics

Axel Oehmichen*, Shubham Jain*, Andrea Gadotti*, Yves-Alexandre de Montjoye
Dept. of Computing and Data Science Institute, Imperial College London
{axelfrancois.oehmichen11, s.jain, gadotti, demontjoye}@imperial.ac.uk

***Abstract*—Mobile phones and other ubiquitous technologies are generating vast amounts of high-resolution location data. This data has been shown to have a great potential for the public good, e.g. to monitor human migration during crises or to predict the spread of epidemic diseases. Location data is, however, considered one of the most sensitive types of data, and a large body of research has shown the limits of traditional data anonymization methods for big data. Privacy concerns have so far strongly limited the use of location data collected by telcos, especially in developing countries.**

In this paper, we introduce OPAL (for OPen ALgorithms), an open-source, scalable, and privacy-preserving platform for location data. At its core, OPAL relies on an open algorithm to extract key aggregated statistics from location data for a wide range of potential use cases. We first discuss how we designed the OPAL platform, building a modular and resilient framework for efficient location analytics. We then describe the layered mechanisms we have put in place to protect privacy and discuss the example of a population density algorithm. We finally evaluate the scalability and extensibility of the platform and discuss related work.

The code will be open-sourced on GitHub upon publication.

I. Introduction

Mobility data – containing accurate user-level trajectories of visited places across long periods – has been extremely valuable for researchers and organizations. Orange's Data for Development (D4D) demonstrated its significant potential for the public good [1]. Mobility data has been used to monitor crime and poverty or assess the impact of natural disasters in real-time [2], [3]. These advancements have been possible thanks to the widespread adoption of mobile phones, including in developing countries where penetration rates vary from 70% in Colombia to 99.9% in Senegal. However, mobility traces are extremely sensitive. In 2009, the Electronic Frontier Foundation listed examples of sensitive information that can be inferred about an individual from their location history [4]. These include the movement of a competitor sales force, attendance of a particular church or an individual's presence in a motel or at an abortion clinic. These legitimate privacy concerns for the potential misuse of mobility data need to be addressed.

Historically, data has been anonymized through de-identification, i.e. the process of transforming personal data to mask the identity of participants. Fully de-identified data is not considered personal data and can be shared or sold without limitations. However, a large body of research has shown that de-identification is not resistant to a wide range of re-identification attacks [5]–[10]. This is especially true for geolocation data, as individual mobility traces are highly unique even among large populations, making them particularly vulnerable to re-identification [7].

OPAL's goal is to unlock the potential of mobility data while preserving privacy. This is achieved by adopting the query-based paradigm for data release: rather than publishing (de-identified) data, OPAL stores the data in a protected environment and allows analysts to send queries about the data. Since analyses are computed using fine-grained data, OPAL makes it possible to achieve better utility and stronger privacy compared to de-identification techniques.

A typical use case for OPAL would be to compute the population density of a certain area for any given time interval, without releasing the full geolocation dataset to the analysts. Ideally, a query-based system such as OPAL would satisfy some important properties:

Secure. The infrastructure is secure against penetration attacks that aim at gaining unauthorized access to data.

Privacy-preserving. The query results should consist of aggregate data, and never disclose individual-level information of users whose records are in the dataset. This guarantee should hold when analysts obtain and combine outputs for multiple queries.

Flexible. Data analysts should be able to submit different queries that serve a large array of statistical purposes. This can be achieved by enabling developers to propose new algorithms that can be loaded on the platform.

Open. The code of the platform and the algorithms should be open-source. This allows for better security, privacy and utility, as everybody can review and contribute to the code and the algorithms.

The contributions of this paper are the following:

- We present OPAL's modular and scalable architecture, designed with a combination of security and privacy features that allows running analysis on the location data while preserving the privacy of the participating users (Section II). We also describe the production deployments in Senegal and Colombia.
- We describe how OPAL's architecture assists in making a density algorithm privacy-preserving. We discuss the implemented privacy mechanism, the utility trade-offs and privacy guarantees (Section III).
- We evaluate the performance of the individual components of the platform as we scale the system to data in the range of billions of records and thousands of queries per second (Section IV).

II. System Overview

In this section, we provide an overview of the multi-layered, modular, and scalable architecture of the OPAL platform. Each layer consists of loosely-coupled micro-services providing flexibility to the framework. The modularity of the architecture and flexibility in the layers allows for horizontal scaling of each layer independently and enables upgrade, addition, and removal of services with zero downtime. The complete platform is designed with four layers - *Endpoints Layer*, *Storage Layer*, *Management Layer*, and *Computation Layer* as shown in Figure 1. At the top, the *Endpoints Layer* provides public APIs to interact with the platform and impart authentication, caching, and auditing services to the platform. All the mobile phone data and platform-specific data (e.g., user details) are saved in the database via the *Storage Layer*. It supports all the other layers by providing replicated, distributed, and scalable storage resources. The *Management Layer* schedules the computation of the analyses on to compute nodes based on their availability and the type of analyses requested. The *Computation Layer* executes the scheduled computations on the scale-out infrastructure which can indifferently be a cluster, a cloud or any other specialized hardware. It ensures that the output for each analysis is privacy-preserving by enforcing the implemented privacy measures.

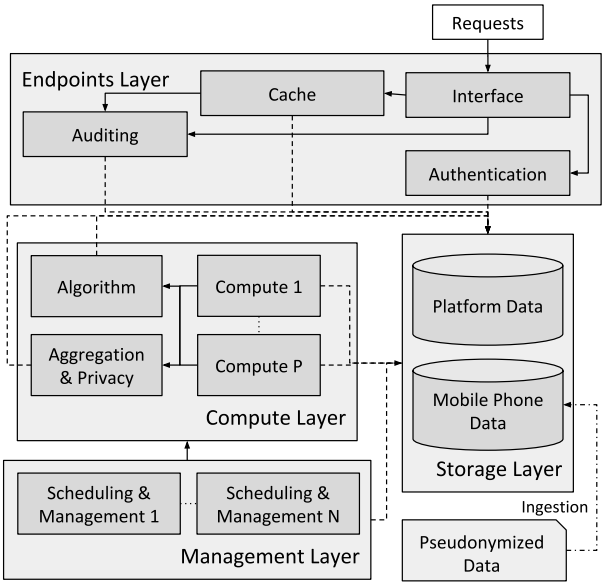

Figure 1. A schematic representation of the architecture of the OPAL platform. Solid lines represent communication between services, dashed lines represent communication between a service and a database.

A. Endpoints Layer

The *Endpoints Layer* provides the only public interface to run analyses over the OPAL data sets. It ensures that only verified and valid requests are processed, all requests are logged, and that the system is responsive at all times. The layer is composed of Interface, Authentication, Cache and Auditing services. Upon successful authentication, if the answer is available with the *Cache service* then it is sent back. Otherwise, the job request is created in the database for the *Management Layer* to schedule.

The *Interface service* provides the public APIs for query and user management. A query is a request for running an analysis against the data available for the requested time interval and an answer is OPAL's output for the requested analysis. Each query must contain the analysis name, start and end date for the data on which it needs to run and other parameters required by the analysis. *Interface service* validates each query by checking that all the required parameters are well-defined.

The *Cache service* stores the answers for all the computed queries with the query parameters and other metadata (date and duration of computation, etc.). They are served directly from the *Cache* if any of those queries are requested again.

The *Auditing service* logs the queries. The log of valid queries must be difficult to modify, even for system administrators. Meanwhile, invalid query logs must be easily accessible to enable periodic analyses to detect possible attacks on the system (see Section III). Thus, the valid queries are logged in an append-only text file, making it harder to modify it without physical access to the system, while the invalid query logs are stored in the database.

B. Storage Layer

The *Storage Layer* manages the mobile phone and platform-specific data. Mobile Phone Data consists of pseudonymized CDRs captured by the telecommunication service provider and the GPS coordinates of antennas. A CDR contains 9 fields (see Table I) that forms the basis for the development of an algorithm on the platform. An antenna is defined by a unique antenna ID, location details (borough, commune, and region) and the presence interval (installation and removal times). Storing installation and removal times is essential for mobile antennas which can be moved from one location to another and have different locations at different times. Analysts can retrieve antenna locations through an API call.

Field	Definition
Timestamp	Datetime at which record is captured by the telecom operator
User ID	Pseudonymized ID of the user whose record was captured
User Country	Country code of the user
Correspond ID	Pseudonymized ID of corresponding user
Correspond Country	Country code for corresponding user
Antenna ID	ID of the antenna the user was connected to during the initiation of interaction
Interact Type	Type of interaction - *call* or *text*
Interact Direction	*Out* if user initiated the interaction, *In* otherwise
Duration	Duration of call in seconds, -1 for text

Table I
Structure of a Call Detail Record (CDR).

The platform-specific data consists of the analyst credentials and permissions, available analyses, query parameters, answers to previously requested queries, status of each micro-service including the available compute nodes, logged invalid requests, and status of each computation (including output and error logs).

The mobile phone data is pseudonymized and ingested periodically into the database by the system administrators. A small to medium-sized country contains billions of records for a year of data [1]. Each computation fetches the data within the requested time interval and can typically range from hundreds to billions of records. The database, for mobile phone data, needs to scale to terabytes of data without significant decreases in performance and provide high-speed data retrieval capability for each concurrent request. However, ingestion speeds can be slower without compromising the overall performance of the platform. Based on these requirements and detailed evaluation (see Section IV) we chose Timescale [11] for storing mobile phone data.

Each service periodically generates significant amount of unstructured data like health status update, query update, etc. MongoDB [12] provides a schema-less data storage with high insertion speed that makes it tailored for our requirements. On top of that, the correctness of our *Scheduling service* relies on the consistency of the data fetched and MongoDB's ability to provide consistency of data across replicas ensure that we can scale without compromising on the correctness of the system.

C. Management Layer

The *Management Layer* is the cornerstone of the OPAL platform. It is responsible for monitoring jobs and compute nodes, and scheduling the job requests for computation. It is crucial for the *Management layer* to have zero downtime, use the new compute nodes as soon as they are available, and to periodically purge the unresponsive compute nodes. To achieve this, we avoid architectures prone to single points of failure. Therefore, unlike master-slave architecture pursued by schedulers such as IBM's LSF [13] or OpenLava [14], OPAL's scheduler is an extension from previous work [15] to provide concurrent multi-master capabilities.

The *Management Layer* is composed of multiple *Scheduling & Management services* running independently in parallel. Each service periodically fetches the jobs and compute nodes from the database for scheduling. As the schedulers run independently, a soft-lock mechanism has been developed to manage the concurrent access to the jobs' and the compute nodes' records in the database. This prevents inconsistencies like the same job getting scheduled multiple times or a single node receiving multiple jobs at the same time. The soft locks are set in the database by the *Scheduler* fetching the records. It periodically fetches the unlocked *IDLE* compute nodes and *QUEUED* jobs. For each fetched job, it checks if a compute node is available to execute it. On a successful check, it tries to set the soft lock on the record and the selected compute node. If successful, a job request is sent to execute the job. The locks are removed once the request is acknowledged by the compute node. This internal concurrency management and the loosely coupled design are the enablers for the multi-master scheduling capabilities of the *Management Layer*.

The *Scheduling & Management service* guarantees that the system works efficiently. To avoid getting bottle-necked with unresponsive jobs or nodes, it flags unresponsive compute nodes and purges failed or stuck jobs. The scheduling of jobs happen each second, purging once a day, and flagging of nodes each minute.

D. Computation Layer

The *Computation Layer* provides execution capabilities to the OPAL platform. Running jobs, generating results, and ensuring the privacy of the users is the role of this layer. It applies a combination of security and privacy features to computations.

OPAL relies on the MapReduce [16] paradigm for defining and executing analysis algorithms (see Section III). Each analysis is a *Map* function that runs on the data of an individual user and a *Reduce* method to aggregate the *Map* results over all the users. For example, a *Map* function could receive a user's CDRs and return the antenna id with most occurrences. A *Reduce* method would then return the count of the occurrences of each antenna id across the *Map* outputs. All algorithms are audited before being added to the platform (see Section III).

The *Map* function runs in a sandboxed environment. This guarantees that the computations run independently over the users (see Section III) by ensuring that the *Map* function interacts with only one user's data in one system process.

Privacy mechanisms mitigate the risk that an attacker can infer details about an individual from an output or a combination of outputs from various analyses on the platform. We provide the functionality to add algorithm-specific privacy modules which help maximize utility and privacy for specific use cases (see Section III).

The *Computation Layer* comprises the *Compute*, the *Algorithm* and the *Aggregation & Privacy* services. The *Algorithm* service manages and versions the analysis algorithms, the *Compute* service fetches data and executes the map functions, the *Aggregation & Privacy* service aggregates the outputs from the *Compute* and applies privacy mechanisms on the aggregated result. Each *Compute* service is associated with a list of type of jobs it can execute, e.g., Python3, R, Spark.

E. Data Flow

The data flow is designed to meet strict privacy requirements. Figure 2 describes the flow of data and the subsequent transformations it undergoes from the raw data to the query output.

The raw data, extracted from the data curator's database, consists of Call Detail Records (CDRs) and antenna details. Records for the users who chose to opt-out are removed and the remaining data is distributed across files. Multiple parallel workers are created for data ingestion. Each worker fetches a file from a shared queue, extracts the country code from the phone number in each record, pseudonymizes the phone numbers, and adds the modified record to a list. Records in the list are ingested in batches. The batch size and number of workers are tuned as per the system configuration. All pseudonymization steps are done using a salt and MD5 [17] hashing function. MD5 has a sufficiently large domain space

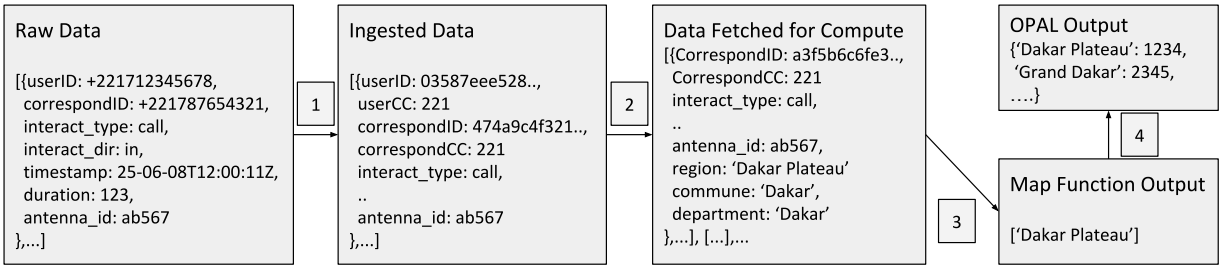

Figure 2. A schematic representation of the flow of the data from the raw data to the OPAL platform's output. 1. Pseudonymizing and ingesting data 2. Data Fetching for compute and creation of user specific CSVs 3. Executing Map function 4. Outputs aggregation and applying privacy mechanisms

to avoid any collision while being small enough to minimize the impact on storage.

The compute nodes create a unique salt for each job execution and the fetched data is pseudonymized again using this salt. This ensures that distinct computations receive data for the same user with different user IDs, making attacks using multiple queries harder to accomplish. Each record is joined with the antenna database to add antenna location to the records. Precaution is taken to ensure that the record timestamp is between the antenna installation and removal time. The fetched data for each unique user is stored in a separate CSV file to sandbox the computations for each user.

F. Development and Deployment

All the micro-services of the OPAL Project follow ECMAScript 8 and use ExpressJS and NodeJS. Docker Compose (in auto-restart mode) is used for the deployment of the services to maximize the platform up-time. It also allows services to be deployed multiple times, across different host machines, for scalability and resilience purposes.

The *Compute* service currently only supports the algorithms written in Python3 and uses Python 3.5 (because of await-/async) to execute them. A new Python library *opalalgorithms* has been developed using Codejail [18] for sandboxing and the Bandicoot [19] toolbox for processing the CDRs of each user.

OPAL has been deployed in production environments in Senegal and Colombia in a two-server setup. The first server hosts the *Endpoints Layer* and the *MongoDB*, while the second server hosts the *Management Layer*, the *Compute Layer* and the *Timescale*. This division minimizes the platform surface exposed to external threats. The break-up of *Storage Layer* across two servers furthermore allows the platform to still serve the answers via the *Cache Service*, in case the second server goes down. Those deployments are actively being used by around a dozen people (from the Senegalese government, United Nations, Agence Nationale de Statistique et de la Démographie, researchers from Orange and Telefònica among others) in each country. These deployments are testimonies to the scalability and robustness of the platform.

III. Privacy of the platform

OPAL belongs to the class of query-based systems, offering data analysts a remote interface to ask questions and receive answers aggregated from many records. Granting access to the data only through queries, without releasing the underlying raw data, mitigates the risk of typical re-identification attacks [5]–[10]. Yet, a malicious analyst can often submit a series of seemingly innocuous queries whose outputs, when combined, will allow them to infer private information about participants in the dataset [20], [21]. In section V, we give an overview of the literature on privacy attacks and privacy-preserving mechanisms. In this section, we describe the different protection layers put in place to protect privacy in query outputs and mitigate the risk of attacks potentially attempted by malicious analysts.

OPAL manages risks using a combination of server-side security, authentication, audit, and network security. However, those protections are only the core layer upon which adopters can rely on and further extend it to meet the most stringent requirements.

Server-side security. Many attacks on privacy employ a relatively large number of queries to circumvent privacy protections, e.g. by averaging out noise [20]. To mitigate this risk, OPAL includes a query rate limitation mechanism (e.g. 100 queries in 7 days). The architecture supports secure execution of algorithms in sandboxed environments through CodeJail. CodeJail (through AppArmor [22]) allows for the creation of a collection of fine-grained rules for limiting the access any executable has over the system. It is designed primarily for Python execution but can be extended to other languages.

Authentication. All requests are verified by the *Authentication service* using OAuth-like tokens. A unique token is associated with each user account and must be supplied through the request. Each user is assigned an access level during their registration. The access level defines the restrictions on the user for algorithms they can access, the limit of spatial and temporal resolution in each query, and other settings defined by the data curator. Upon successful authentication, the *Interface service* validates the request. Further restrictions can also be implemented such as maximum sampling size for an analysis.

Audit. The *Auditing service* provides access to the valid and the invalid requests made to the system. Auditing is an important part of the security of the platform as it enables system administrators and governance board members for ethical oversight to review all previous queries and detect any potentially suspicious request or sequence of requests.

Network security. The layers are deployed into two differ-

ent VLANs to shield the platform from external brute force attacks. This siloing enables to expose only the *Interface service* in the *Endpoints Layer* to client's applications while the data and services are safely hidden from the rest of the network. The platform is hosted behind a firewall and only the *Interface service* is exposed to the internet.

MapReduce. OPAL relies on the MapReduce [16] paradigm for computation. The Reduce methods currently supported are *count*, *sum* and *median*. The use of MapReduce helps privacy in two ways. First, it is easier to audit algorithms, since the Map function is applied independently to each user in the dataset, and hence it makes it hard for an attacker to hide conditions that try to re-identify specific records. For example, an IF statement that checks whether the user made a call with duration 5m23s on 10/07/2018 would look suspicious and raise a flag in the auditing phase. Second, the MapReduce paradigm ensures that every output is the result of a final aggregation step (i.e. the Reduce method). For example, if the *count* function is selected, this ensures that every user can contribute by at most 1 to the final output. While this is not enough to guarantee privacy, it offers a first layer of protection and simplifies the design of additional privacy-preserving measures.

Algorithm auditing. All algorithms are evaluated by a committee before being installed on the platform. Only system administrators can install an algorithm. Further, if an algorithm needs to use its own privacy module (see below), a special token has to be passed in the request body that is verified against the token in the *Algorithm* service configuration file. This token is made available only to algorithm auditors and, hence, it ensures that no algorithm with custom privacy module is added without being audited.

Privacy module. Every OPAL algorithm consists of two components:

1) The *analysis algorithm*, composed of the Map function and Reduce method (see Section II).
2) An optional *privacy module* providing privacy-mechanisms for the query.

The privacy module receives the output of the analysis algorithm and has to ensure that the final output of each query does not disclose personal data. This is typically achieved via noise addition, query set size restriction, and other techniques. The privacy module can provide differential privacy [23] or any other privacy protection that the developer wants to implement. Although solutions for general-purpose privacy-preserving data analytics have been proposed [24]–[29], they present limitations for utility, flexibility, or privacy [21], [29], [30]. Algorithm-specific techniques can give strong privacy protections and yield accurate results but need to be designed and tuned for each new algorithm. OPAL allows every algorithm to include a privacy module specific to that algorithm, allowing developers to achieve a better privacy/utility tradeoff in their algorithms. In this section, we present a privacy module for the density algorithm that gives good privacy protections and provides good utility. In particular, the algorithm enforces geo-indistinguishability (GI) [31], a variant of differential privacy (DP) [23].

A. The population density algorithm

The density algorithm is used to release the number of users who spent most of their time in a certain area in a given time interval. The algorithm accepts five parameters from the analyst: *resolution*, *keySelector*, *startDate*, *endDate*, *sample*.

The keySelector is a list of areas for which the density needs to be computed. This could be, for example, the id of a specific borough or the name of a city. The resolution parameter selects the spatial resolution of the requested locations. There are three different resolution levels: *borough*, *commune* and *region*. The list of all available areas and corresponding resolution level is made available in the API documentation.

The startDate and endDate parameters specify the time interval of interest. Both parameters can select any day and any time of the form `hh:00:00`. The sample parameter is a value between 0 and 1 that specifies the (random) fraction of users sampled by the algorithm to compute the query. There are three available values: $0.01, 0.1, 1$. Larger sample parameters yield better accuracy but require more time to compute, as the platform needs to process more users. In density, sampling is not used to improve privacy guarantees.

The output of the density query is a list of $(key, value)$ pairs, where each key is one of the areas specified in keySelector and value represents the number of users that spent most of their time in that area during the specified time interval. Note that a user might visit multiple locations in the same time interval, but our algorithm adopts a *winner-takes-all* approach, i.e. it assigns each user to at most one area (where the user spent most of their time in the requested interval). A user can thus contribute to count of only one element in the keySelector.

We denote by $\mathsf{density}(L, T_1, T_2, \rho)$ the output of the density algorithm for the location L in the interval $[T_1, T_2]$ with sampling parameter ρ. To simplify the exposition, in the rest of this section we present the details of density when a single location L is selected. L denotes a generic geographic area for an arbitrary resolution, and we denote by $\mathcal{L}$ the set of locations at the resolution of L. So, for example, $\mathcal{L}$ could be a set of cities or a set of regions. If keySelector contains a list of locations $(L_1, \ldots, L_n)$ that belong to $\mathcal{L}$, the density algorithm (including its privacy module) is run independently on each L_i.

B. Privacy module for density

We now present our design of the privacy module for the density query. This serves as an example to demonstrate the flexibility of the OPAL platform and offers a starting point for the development of other algorithms.

The density algorithm employs three layers of protection: geo-indistinguishability, low-count suppression, and output noise addition. The specific combination and implementation of these layers integrate flawlessly with OPAL's infrastructure and distributed computation flow.

Geo-indistinguishability. GI is a formal notion of location privacy [31] to obfuscate single user locations. The formal definition of GI depends on a parameter ε that controls how quickly the privacy guarantees vanish for points that are far from the true location. The parameter ε is called *privacy loss*.

Definition 1 (ε-geo-indistinguishability): Let $\mathcal{X}$ be a set of locations and let $\mathcal{A}\colon \mathcal{X} \to \mathcal{X}$ be a randomized algorithm. Denote by $d(\cdot,\cdot)$ the Euclidean distance. $\mathcal{A}$ satisfies *ε-geo-indistinguishability* if, for all $x, x' \in \mathcal{X}$ and $S \subseteq \mathcal{X}$,

$$\Pr[\mathcal{A}(x) \in S] \leq e^{\varepsilon d(x,x')} \Pr[\mathcal{A}(x') \in S].$$

Definition 2 (Planar Laplace distribution): Let $\varepsilon \in \mathbb{R}^+$ and $x \in \mathbb{R}^2$. The planar Laplace distribution centered at x is the probability distribution on $\mathbb{R}^2$ with pdf

$$D_\varepsilon(x)(x') = \frac{\varepsilon^2}{2\pi} e^{-\varepsilon d(x_0,x)}.$$

Consider the mechanism that, on each input x, outputs x' drawn from $D_\varepsilon(x)(x')$. One can prove that this mechanism satisfies ε-GI. In practice, the obfuscated location x' is obtained by adding to the true location a noise value sampled from a planar Laplace distribution centered in zero (see [31]).

In density, the location associated with every CDR is obfuscated using planar Laplace noise with parameter ε, producing a sanitized dataset (this method is discussed in [32]). In our implementation, we sanitize the data on-the-fly, using pseudo-random generators, to avoid the need to store a sanitized copy of the dataset and give more flexibility to developers (see next paragraphs). Nevertheless, to simplify the exposition, sometimes we use the expression "sanitized dataset" to refer to the dataset that we would obtain if we saved every user location obfuscated while executing the algorithm. By default, every algorithm uses this sanitized dataset, but can also request access to the original dataset.

Low-count suppression. The algorithm counts how many users spent most of their time in the selected area. The count is computed on the dataset sanitized with GI. If the count for a certain area is below a fixed threshold B, then the count is suppressed and the value associated with that area is set to `ValueTooLow` in the output.

Output noise addition. Finally, density adds random noise to every count that is not suppressed. The random noise value is drawn from a normal distribution $\mathcal{N}(0, \sigma^2)$, and is sampled using a pseudo-random number generator. The seed is set to:

$$seed = \text{hash}(L, T_1, T_2, \rho, \texttt{salt}),$$

where `salt` is a long string known only to the data curator but accessible from any algorithm, and the default hash function is SHA-512. This ensures that the noise value is the same for the selected area and time interval, and hence it cannot be averaged out by repeating the same query [28].

The full algorithm is presented in detail in Procedure density. In the algorithm, the *majority element* (or *location*) of a list is the element with most occurrences. If there is a tie, then the majority element is selected at random among the most frequent.

Procedure density(L, T_1, T_2, ρ; ε, B, σ)

Input: Defined by analyst: location $L \in \mathcal{L}$, time start T_1, time end T_2, sampling parameter ρ.
Defined by data curator: GI parameter ε, minimum threshold B, noise stdev σ

Output: number of sampled users who spent most of their time in L during $[T_1, T_2]$

$\mathcal{D} \leftarrow$ fraction ρ of total users, selected at random
$density \leftarrow 0$
for $i \leftarrow 1$ **to** $|\mathcal{D}|$ **do**
  $\{x_1, \ldots, x_n\} \leftarrow$ locations with timestamps of $user_i$ between T_1 and T_2
  for $j \leftarrow 1$ **to** n **do**
    $seed \leftarrow \text{hash}(user_i, x_j.time, x_j.loc, \texttt{salt})$
    $x'_j \leftarrow$ random location drawn from planar Laplace pdf $D_\varepsilon(x_j)(\cdot)$ seeded with $seed$
    $l_j \leftarrow$ nearest location from x'_j in $\mathcal{L}$
  For each 10min interval in $[T_1, T_2]$ select the majority location L_j for that interval
  $\{L_1, \ldots, L_m\} \leftarrow$ majority locations, one for each 10min interval
  $L^* \leftarrow$ majority location of $\{L_1, \ldots, L_m\}$
  if $L^* = L$ **then**
    $density \leftarrow density + 1$
if $density < B$ **then**
  return `ValueTooLow`
else
  $seed \leftarrow \text{hash}(L, T_1, T_2, \rho, \texttt{salt})$
  $noise \leftarrow$ draw random value from $\mathcal{N}(0, \sigma^2)$ seeded with $seed$
  return $density + noise$

Choice of privacy parameters. The `density` algorithm depends on three parameters: ε, B and σ. As these parameters determine the privacy protections of the mechanism, they must be fixed by the data curator. The defaults are:

- $\varepsilon = 10\,\text{km}^{-1}$. This ensures that the obfuscated locations are statistically indistinguishable with confidence ε from all the other locations within distance d. For the exact meaning of this guarantee, we refer to [31]. In particular, the expected distance between the true and obfuscated location is $2/\varepsilon$.
- $B = 50$. We believe that such a high value does not affect utility significantly, as data analysts are not generally interested in precise density values for populations smaller than 50 individuals.
- $\sigma = 10$. Seen as an application of the Gaussian mechanism [23], this corresponds to the protection level guaranteed by (ξ, δ)-differential with privacy loss $\xi = 0.6$ and parameter $\delta = 2.5 \cdot 10^{-7}$ *for a single query*, which is generally believed to provide meaningful privacy [33]. Note that `density` does not really enforce DP in general, as we do not limit the privacy budget over multiple queries. We discuss this more in detail later on.

The utility of `density`. The `density` algorithm

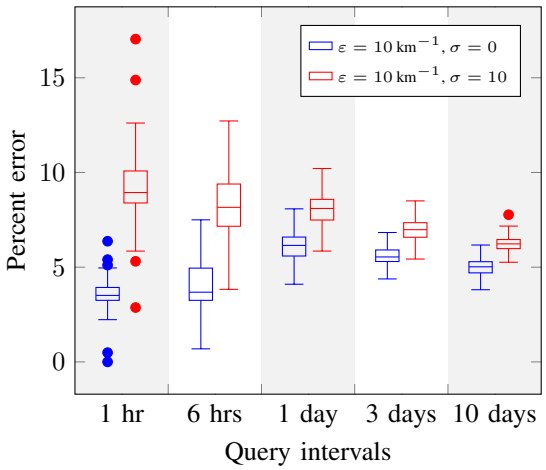

Figure 3. Percent error for queries computed using different time interval lengths. In blue: Only geo-indistinguishability is applied (no output noise addition). In red: Both geo-indistinguishability and output noise addition are applied.

presents several advantages for utility, especially with respect to accuracy of the outputs and running time. We tested `density` on a real mobility dataset with millions of users and thousands of antennas, using antenna-level resolution and setting $\varepsilon = 10\,\mathrm{km}^{-1}$. We ran `density` across all antennas over 100 different time intervals of length 1 hour, 6 hours, 1 day, 3 days and 10 days. When $\sigma = 0$ (i.e. there is no output noise addition), the median relative error between `density` outputs larger than $B = 50$ and the true density values (i.e. without defenses) is between 3-6% for all different lengths. Setting the default $\sigma = 10$, the median relative error goes up to 6-9%. This is because, even for longer time intervals, a significant fraction of the outputs are rather small, but the parameter σ for the Gaussian noise is not scaled to the output. This is needed to provide stronger privacy guarantees [34]. As for the computational efficiency, the average overhead due to the GI sanitization is only 3%.

The privacy guarantees of `density`. The `density` algorithm relies on GI as a formal notion of privacy. The specific choice of GI as an obfuscation method is due, among other things, to some of its mathematical properties. Most notably, it abstracts from the adversary's background knowledge and the total privacy loss grows naturally (linearly) with the number of observed user locations [31]. Additionally, GI's guarantees are preserved under multiple queries and specifically, they are compatible with sampling: running the same query with different sampling parameters does not affect the privacy protection provided by GI.

Although inspired by DP, GI is fundamentally different. It is easy to check that simply releasing aggregate statistics computed on a dataset sanitized with GI does not enforce DP. Similarly, the `density` algorithm is not designed to enforce DP. Although the addition of Gaussian noise to each query output ensures (ξ, δ)-DP on that output, the total (theoretical) privacy loss for DP increases linearly with the number of queries. In the current implementation of `density`, we do not limit the privacy budget, hence the theoretical total privacy loss for DP is unbounded. However, the overall rate of queries per analyst is limited by OPAL's *Interface service*. The default limit is 50 queries/week.

Applying DP to mobility data provides very strong guarantees, but it is extremely challenging to preserve utility and flexibility. One example is the mechanism by Acs et al. [35] to release the population density in Paris. While the proposed solution enforces DP, it presents two important limitations that make it impractical for our use case. First, the data is available only for one week. This allows to pre-sample a limited number of locations for each user, hence improving privacy while preserving good utility. In contrast, our algorithm can be used to query data that spans several months or even years. Second, the density is computed for slots of one hour. To obtain the density across larger time frames, one would then take the sum over the selected one-hour slots. However, this leads to biased estimates for larger time intervals, as the same user may be counted multiple times in different slots.

Attacks. We test the the privacy protections of `density` against the membership inference attack (MIA) by Pyrgelis et al. [36], [37]. This is a state-of-the-art membership attack on aggregate mobility data: the attacker receives the outputs of some density queries, and her goal is to determine whether a certain user's data was used to compute the outputs. MIA assumes that the attacker has some auxiliary information. In our experiments we use the subset of location prior. Due to space constraints, we refer the reader to [37] for the attack details.

We run the attack in a setting similar to [37] (using our implementation, based on the original one). We consider 4 weeks of data for a city, and remove all the users with less than 10 recorded activities across 4 weeks, obtaining a dataset with 260k users. We note that MIA is less effective on users with lower activity, thus removing them *increases* the average success rate of the attack. Following the choice of parameters in the original paper, we assume that the attacker has access to 20% of the dataset ($\alpha = 0.2$) and we run MIA on three groups of 50 targets each, sampled from the *highly*, *mildly*, and *somewhat* mobile groups respectively. We investigate the results for several aggregation sizes (m), choosing logistic regression as classifier as it gives the best results.

Pyrgelis et al. measure MIA's effectiveness with a target-specific AUC score. In Figure 4 we report the average AUC over the 150 targets when the attack is run against `density` (using default privacy parameters) and against an algorithm that releases the true density values (without defenses). Following Pyrgelis et al., we additionally measure the robustness of `density` by computing the *privacy gain* (PG). This metric measures the normalized difference in the attack's performance when run against `density` and when run against an algorithm that releases the true density values. The privacy gain can vary between 0 (the attack attains the same effectiveness with or without defenses) and 1 (the attack is completely ineffective against defenses). We report the results in Figure 5 and find that `density` protects against MIA. The average privacy gain is almost 1 for every large aggregation size implying that `density` provides strong privacy protection to almost

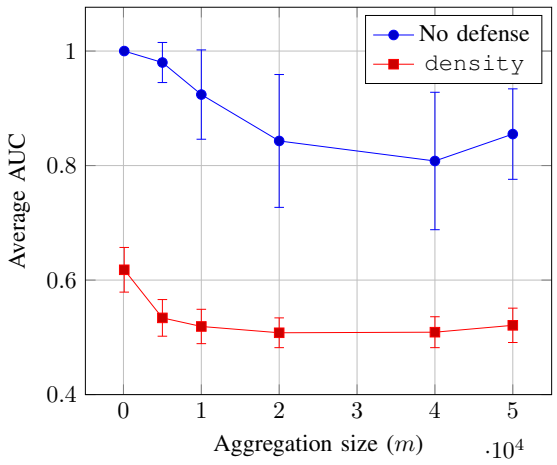

Figure 4. Average AUC for MIA for various aggregation sizes against no defense and `density`.

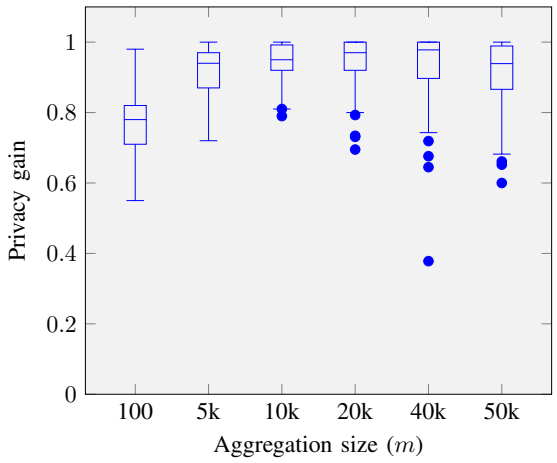

Figure 5. Privacy gain of `density` for various aggregation sizes.

every target. We note that the privacy gain slightly decreases for larger aggregation sizes. This is due to some "artificial" regularity that improves the effectiveness of the attack when run on larger aggregation sizes. We refer to [36] for the details.

Naturally, these results do not guarantee that `density` is robust against any possible attack. However, we believe that such a high resilience to the state-of-the-art attack on aggregate mobility data is a good indication that `density` protects individual's privacy well.

IV. Scalability of the Platform

In this section, we study the performance of the individual layers as the platform scales.

A. Management Layer

We evaluated the performance and resilience of the *Management Layer* as the system scales. For both evaluations, a single instance of MongoDB was deployed with 1000 concurrent compute nodes. Requests are submitted to execute a sleep job of 1s. Each job is scheduled, computed, and the results are stored in MongoDB. A large number of compute nodes

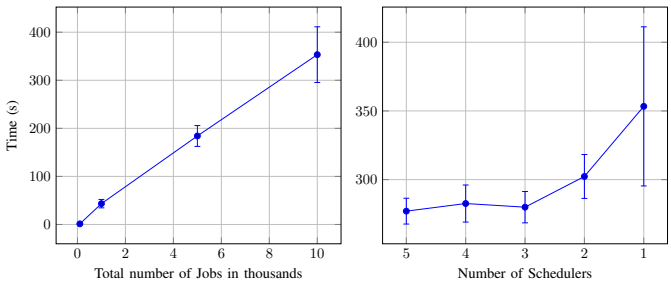

Figure 6. Left: Performance of a single scheduler w.r.t. the submission size. Each point represents the average running time of 10 experiments along with the standard deviation. Right: Performance of the *Management Layer* as the number of schedulers decreases. Each point is the average running time of 3 experiments along with the standard deviation.

ensure that the time measured truly evaluates performance of the *Management Layer* and not the computation bottleneck. All the experiments were run on a single machine (48 cores, 250GB RAM, 256GB storage RAID 1 @7200RPM).

To evaluate the scaling capacity of the *Managament Layer* as the number of concurrent jobs increases, we submit N batches (N $= 1, 10, 50$ and 100) with 100 jobs per batch to a single scheduler. A scenario of more than 10k jobs is unlikely in a production environment as each job takes time (in hours) to process. Figure 6 left plot shows that that submission and scheduling scales linearly with a large number of requests.

To evaluate the resilience of the *Management Layer* as the system horizontally scales, we submit a total of 10k jobs with a varying number of schedulers. Five independent clients (batches of size 100) insert jobs in MongoDB with M schedulers (M=5,..,1) running in the background. Figure 6 right plot shows that computation time increases slightly as the number of schedulers decreases. This behavior re-asserts the resilience of our *Management Layer*, enabled by its multi-master design, by showing that failure of multiple schedulers has a negligible effect on performance.

B. Storage Layer

A key requirement of the platform is to be able to store and serve billions of CDRs efficiently (see Section II-B). We benchmarked various potential existing solutions, and evaluated the performance of the chosen database with the scale of data. We shortlisted four candidates after thorough review: MongoDB, Timescale, InfluxDB, and Druid. To evaluate the candidates, we conducted following benchmarks:

1) Insert a month of CDRs (74GB, this is typical size for a month of data in a small to medium size country, stored in a single CSV file) in a single process with batch size 10k. Evaluated time is average of 2 runs.
2) Run 5 different select queries fetching records in random 5 min intervals. Each retrieval fetched 60k records on average. Average time over all the queries is reported.

Each solution was deployed in a container and the benchmarks were executed sequentially using two identical machines (24 cores, 100GB RAM, 7200RPM storage), one hosted the containers while the other one executed the scripts.

Databases	Insertion time	Select Time
MongoDB	13h	34m
Timescale	46h	0.7s
InfluxDB	34h	7s
Druid	>48h	16m

Table II
BENCHMARKS FOR THE FOUR POTENTIAL DATABASE SOLUTIONS.

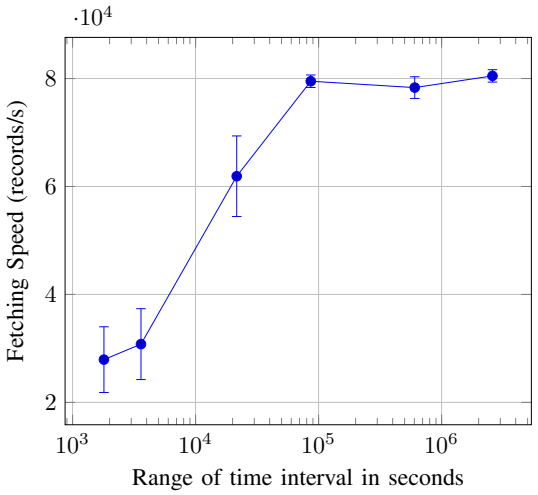

Figure 7. Performance of select queries for Timescale w.r.t time interval of the query. The fetching overheads are significant for smaller queries but become marginal as the query interval size increases.

From Table II, we can see that Timescale provides an order of magnitude lesser select time compared with the other solutions while still having acceptable performances for insertion time. The select is of key importance for OPAL as it allows for faster computations of the requests in a production environment. It is for these reasons that we chose Timescale as the database. Timescale also provides us with standard SQL engine and the extensibility capabilities of PostgreSQL.

Further, we evaluated the performance of Timescale in a single deployment instance as the scale of data increases. All the experiments were performed with Timescale-0.11, Postgres-10, Python 3.5 and asyncpg [39] deployed in a container on a single machine (48 cores, 189 GB RAM, 8 TB storage RAID 5 @10k RPM). We had 6 months of data with more than 8 billion unique records.

First, we evaluated the insertion speeds (total data inserted divided by the total time taken) to store an increasing number of records in the database. The raw data was stored in compressed CSV files, each containing an hour of data. Eight workers ran in parallel, each retrieving a CSV file from a shared queue and pseudonymizing each record before inserting them in the database in batches of 2 million. We found that insertion speed remained stable ∼13MB/s (average over 3 runs) as the amount of data inserted increased from 0 to 973 GB. This can be attributed to the transparent time-space partitioning provided by Timescale [11]. Overall, 6 months of data was inserted in less than a day.

Then, we evaluated the selection speed as the data is fetched for increasing time intervals of 30 mins., 1 hr., 6 hrs., 1 day, 7 days, and 30 days. For each interval we measure the time taken

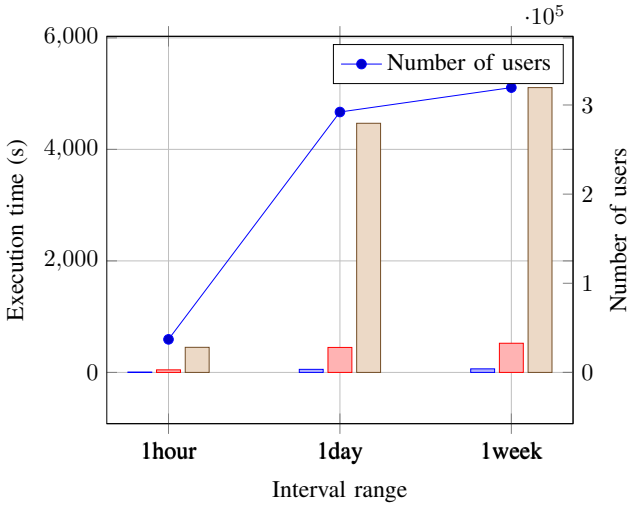

Figure 8. Performance of the *Compute layer* for queries with various interval range sizes and sampling parameters (blue: 1%, red: 10%, brown: 100%). We also plot the number of users for each interval range.

from sending a query to completely parse the result and the number of records fetched, averaged over 20 queries. Figure 7 shows speed remains largely similar as the intervals go from 1 day to 1 month. The lower speeds for smaller interval lengths can be attributed to the database overheads which become negligible as the interval length increases.

C. Computation Layer

We evaluate the scalability of the *Computation Layer* to process large amounts of data using Codejail sandbox. This experiment was conducted with two weeks of data containing over 44 million records for 320k users on a single machine (8 cores, 64 GB RAM, 7200 RPM storage). We measured the time to compute density for 3 different intervals (1 hour, 1 day and 1 week) over 3 different user sampling parameters (1%, 10%, and 100%). For each computation, the data is fetched in batches of 50k users, and the map algorithm runs in parallel over each fetched user. We used 6 workers for user processing and 1 for fetching the data.

Figure 8 shows that the increase in compute time for intervals of 1 hour to 1 day to 1 week is directly proportional to the number of users in that interval. This behavior is attributed to the fact that sandboxing is the bottleneck in the computation. Sandboxing requires each user data to be saved as a distinct file, and a new process is spawned for processing data of each user, making it IO and CPU intensive. It takes less than 1.5hrs to compute density for all the users for a week of data while the analysis on 10% of the users over the same period requires less than 9 minutes. Sampling is thus tailored for scenarios when quick answers are required and utility can be slightly compromised.

V. RELATED WORK

A large range of attacks on query-based systems have been developed since the late 70's [40], [41]. Most of these attacks show how to circumvent privacy safeguards (e.g. query set

size restriction and noise addition) in specific setups. In 2003, Dinur et al. [42] proposed the first example of an attack that works on a large class of query-based systems. Since then, numerous other attacks have been proposed in the literature. These attacks address different limitations of previous ones, particularly the computational time required to perform them. A recent survey from Dwork et al. [43] gives a detailed overview of attacks on query-based systems.

Privacy research has been increasingly focused on providing provable privacy guarantees to defend query-based systems against such attacks. However, the development of a privacy-preserving platform for general-purpose analytics is still an open problem [29]. General-purpose analytics usually refers to systems that allow analysts to send many queries of different type, using a rich and flexible query language. Some solutions based on differential privacy have been proposed, the main ones being PINQ [24], wPINQ [25], Airavat [26], and GUPT [27]. All of these systems however present limitations [29], e.g. in simplicity of use for the analyst, which must provide additional query parameters to the differential privacy implementation. In particular, Airavat is based on MapReduce like OPAL and enforces differential privacy by using a simple application of the Laplace mechanism [23]. Like other general differentially private mechanisms, a straightforward application of the Laplace mechanism often destroys the utility of the data for multiple queries [34]. Specifically, every aggregation method supported by OPAL (count, sum, median) could be easily made differentially private with the standard Laplace or exponential mechanisms [23], [44], but this solution would require to add a lot of noise to outputs in order to provide meaningful guarantees.

In 2017, Johnson et al. [29] proposed a framework for general-purpose analytics, called FLEX, which enforces differential privacy for SQL queries without requiring any knowledge about differential privacy from the analyst. However, the actual utility achieved – level of noise added – by FLEX has been questioned [30]. Diffix, a patented commercial solution that acts as an SQL proxy between the analysts and a protected database [28], [45] has recently been proposed as an alternative to differential privacy. However, Diffix's anonymization mechanism has been shown to be vulnerable to some re-identification attacks [21], [46], [47].

VI. Discussion and future work

The current implementation presents few challenges.

System. There is currently no live ingestion of new data, as the data gets loaded periodically in bulk. This limits the platform's capabilities to, e.g., monitor a crisis and provide adequate information for search and rescue parties.

The current implementation is targeted to scale up to the medium-sized countries (e.g. up to 50M people). In order to scale up to the larger countries, further work would be required on the database side to ensure the efficient storage and retrieval of the data across the different workers and further improve the computation performances in the sandboxed environment.

Privacy. A more generic approach to privacy on the platform is another challenge. Currently, generic noise addition, strict caching and query set size restriction offer only a basic layer of protection. Privacy is mainly provided by the privacy module, which is algorithm-specific. General privacy-preserving mechanisms that apply automatically to the outputs of any query could simplify the development of new algorithms. However, preserving good utility for general-purpose analytics is still an open research problem (see section V).

In the future, we plan to test the robustness of `density` against other attacks from the literature, profiling attacks [48] and trajectory recovery attacks [49]. Finally, more investigation is required to assess the viability of GI for algorithms other than density, from both the privacy and utility perspectives.

VII. Conclusion

In this paper, we presented OPAL's architecture for the efficient and scalable analysis of massive amounts of location data in a secure and privacy-preserving fashion. The production deployments in Senegal and Columbia show that the OPAL platform provides a flexible, scalable and robust solution for large scale data analysis in the context of location data analysis. The algorithms provided as part of the platform aim at providing a privacy compliant solution while retaining a high utility owing to the different privacy mechanisms that have been put in place.

Acknowledgements

The authors would like to thank Imperial College London's Computational Privacy Group as well as Andrew Young, Emmanuel Letouzé, Carlos Mazariegos, Nicolas De Cordes, Thomas Heinis. We acknowledge support from Agence Française de Développement as part of its financial assistance to the OPAL project and the eTRIKS project. The opinions expressed in this article are the authors' own and do not reflect the view of the Agence Française de Développement.

References

[1] Y.-A. de Montjoye, Z. Smoreda, R. Trinquart, C. Ziemlicki, and V. D. Blondel, "D4d-senegal: the second mobile phone data for development challenge," *arXiv preprint arXiv:1407.4885*, 2014.

[2] R. Wilson, E. zu Erbach-Schoenberg, M. Albert, D. Power, S. Tudge, M. Gonzalez, S. Guthrie, H. Chamberlain, C. Brooks, C. Hughes *et al.*, "Rapid and near real-time assessments of population displacement using mobile phone data following disasters: the 2015 nepal earthquake," *PLoS currents*, vol. 8, 2016.

[3] L. Bengtsson, J. Gaudart, X. Lu, S. Moore, E. Wetter, K. Sallah, S. Rebaudet, and R. Piarroux, "Using mobile phone data to predict the spatial spread of cholera," *Scientific reports*, vol. 5, p. 8923, 2015.

[4] A. J. Blumberg and P. Eckersley, "On locational privacy, and how to avoid losing it forever," *Electronic frontier foundation*, vol. 10, no. 11, 2009.

[5] L. Sweeney, "Weaving technology and policy together to maintain confidentiality," *J. Law Med. Ethics*, 1997.

[6] A. Narayanan and V. Shmatikov, "Robust de-anonymization of large sparse datasets," in *IEEE Symposium on Security and Privacy*, 2008.

[7] Y.-A. de Montjoye, C. A. Hidalgo, M. Verleysen, and V. D. Blondel, "Unique in the Crowd: The privacy bounds of human mobility," *Scientific Reports*, 2013.

[8] Y.-A. de Montjoye, L. Radaelli, and et al., "Unique in the shopping mall: On the reidentifiability of credit card metadata," *Science*, 2015.

[9] C. Culnane, B. I. P. Rubinstein, and V. Teague, "Health data in an open world," *ArXiv e-prints*, Dec. 2017.

[10] P. Ohm, "Broken promises of privacy: Responding to the surprising failure of anonymization," *UCLA Law Rev.*, 2010.

[11] "Timescale: Open-source time-series database powered by postgresql," www.timescale.com/.

[12] "mongoDB," www.mongodb.com/.

[13] IBM, "IBM Platform LSF V9.1.3 documentation." [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSETD4_9.1.3/lsf_welcome.html

[14] P. Joshi and M. R. Babu, "Openlava: An open source scheduler for high performance computing," in *International Conference on Research Advances in Integrated Navigation Systems*, 2016.

[15] A. Oehmichen, F. Guitton, and et al., "eTRIKS analytical environment: A modular high performance framework for medical data analysis," *2017 IEEE Big Data*, 2017.

[16] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, 2008.

[17] R. Rivest, "The MD5 Message-Digest Algorithm RFC 1321," *arXiv*, 1992.

[18] "GitHub - edx/codejail: Secure code execution." [Online]. Available: https://github.com/edx/codejail

[19] Y.-A. de Montjoye, L. Rocher, and A. S. Pentland, "bandicoot: a Python Toolbox for Mobile Phone Metadata," *Journal of Machine Learning Research*, 2016.

[20] C. Dwork, A. Smith, T. Steinke, and J. Ullman, "Exposed! A Survey of Attacks on Private Data," *Annual Review of Statistics and Its Application*, 2017.

[21] A. Gadotti, F. Houssiau, L. Rocher, B. Livshitsa *et al.*, "When the signal is in the noise: Exploiting diffixs sticky noise," in *USENIX Security Conference Proceedings*, 2019.

[22] "Apparmor application security for linux." [Online]. Available: https://wiki.ubuntu.com/AppArmor

[23] C. Dwork, F. McSherry, K. Nissim, and A. Smith, *Calibrating Noise to Sensitivity in Private Data Analysis*. Springer, 2006.

[24] F. D. McSherry, "Privacy integrated queries: An extensible platform for privacy-preserving data analysis," in *Proceedings of the 2009 ACM SIGMOD*, 2009.

[25] D. Proserpio, S. Goldberg, and F. McSherry, "Calibrating data to sensitivity in private data analysis: A platform for differentially-private analysis of weighted datasets," *Proceedings VLDB Endowment*, 2014.

[26] I. Roy, S. T. V. Setty, and et al., "Airavat: Security and privacy for MapReduce," in *NSDI*, 2010.

[27] P. Mohan and T. et al., "GUPT: Privacy preserving data analysis made easy," in *Proceedings of the 2012 ACM SIGMOD*, 2012.

[28] P. Francis, S. Probst Eide, and R. Munz, "Diffix: High-Utility database anonymization," in *Privacy Technologies and Policy*. Springer International Publishing, 2017.

[29] N. Johnson, J. P. Near, and D. Song, "Towards practical differential privacy for sql queries," *ArXiv e-prints*, 2017.

[30] F. McSherry, "Uber's differential privacy .. probably isn't," https://github.com/frankmcsherry/blog/blob/master/posts/2018-02-25.md, 2018.

[31] M. E. Andrés, N. E. Bordenabe, and et al., "Geo-indistinguishability: Differential Privacy for Location-based Systems." ACM, 2013.

[32] N. E. Bordenabe, "Measuring privacy with distinguishability metrics: Definitions, mechanisms and application to location privacy," Ph.D. dissertation, cole Polytechnique, 2014.

[33] C. Dwork and A. Smith, "Differential privacy for statistics: What we know and what we want to learn," *Journal of Privacy and Confidentiality*, 2010.

[34] C. Dwork and A. Roth, "The Algorithmic Foundations of Differential Privacy," *Foundations and Trends in Theoretical Computer Science*, 2013.

[35] G. Acs and C. Castelluccia, "A Case Study: Privacy Preserving Release of Spatio-temporal Density in Paris," in *KDD '14 Proc. of the 20th ACM SIGKDD int. conf.*, Aug. 2014.

[36] A. Pyrgelis, C. Troncoso, and E. D. Cristofaro, "Knock Knock, Who's There? Membership Inference on Aggregate Location Data," in *Proc. of NDSS 2018 Conf.*, 2018.

[37] A. Pyrgelis, C. Troncoso, and E. De Cristofaro, "Under the hood of membership inference attacks on aggregate location time-series," *arXiv preprint arXiv:1902.07456*, 2019.

[38] S. Oya, C. Troncoso, and F. Pérez-González, "Is Geo-Indistinguishability What You Are Looking for?" *arXiv:1709.06318 [cs]*, Sep. 2017.

[39] Python Software Foundation, "A fast PostgreSQL Database Client Library for Python/asyncio." [Online]. Available: https://github.com/MagicStack/asyncpg

[40] D. E. Denning, "Are statistical data bases secure," in *Proc. AFIPS*, 1978.

[41] L. Beck, "A security machanism for statistical database," *ACM Trans. Database Syst.*, 1980.

[42] I. Dinur and K. Nissim, "Revealing information while preserving privacy," in *ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2003.

[43] C. Dwork, A. Smith, T. Steinke, and J. Ullman, "Exposed! a survey of attacks on private data," *Annual Review of Statistics and Its Application*, 2017.

[44] F. McSherry and K. Talwar, "Mechanism Design via Differential Privacy," in *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS'07)*, 2007.

[45] P. Francis, S. Probst-Eide, P. Obrok, C. Berneanu, S. Juric, and R. Munz, "Extended Diffix," *ArXiv e-prints*, Jun. 2018.

[46] A. Cohen and K. Nissim, "Linear Program Reconstruction in Practice," *TPDP 2018*, Oct. 2018.

[47] A. Pyrgelis, C. Troncoso, and E. De Cristofaro, "On location, time, and membership: Studying how aggregate location data can harm users' privacy," https://www.benthamsgaze.org/2018/10/02/on-location-time-and-membership-studying-how-aggregate-location-data-can-harm-users-privacy/, Oct. 2018.

[48] ——, "What Does The Crowd Say About You? Evaluating Aggregation-based Location Privacy," *Proceedings on Privacy Enhancing Technologies*, 2017.

[49] F. Xu, Z. Tu, and et al., "Trajectory Recovery From Ash: User Privacy Is NOT Preserved in Aggregated Mobility Data," in *Proc. of WWW '17 26th Int. World Wide Web Conf.*, 2017.