User Private Clouds

Nicholas Riasanovsky

User Private Clouds

by

Nicholas J. Riasanovsky

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Computer Science

in

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor David E. Culler, Chair
Professor Joseph M. Hellerstein

Spring 2020

User Private Clouds

by Nicholas J. Riasanovsky

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

[illegible]

Professor David E. Culler
Research Advisor

5/26/2020

(Date)

* * * * * * *

[illegible]

Professor Joseph M. Hellerstein
Second Reader

5/27/2020

(Date)

Abstract

User Private Clouds

by

Nicholas J. Riasanovsky

Master of Computer Science

University of California, Berkeley

Professor David E. Culler, Chair

Applications that collect data from their participants often give users minimal control over their own data, although they sometimes allow users to request that their data be used properly. In contrast, User Private Clouds (UPC) let users dictate a unique set of services that run in the cloud and are permitted to process their data. In UPC, users have sole ownership of their data, but they can still participate in large scale applications because the UPC trust model enables safe use of global differential privacy. This work demonstrates UPC by providing a sample implementation of the cloud-based layer. We demonstrate writing software for it by adapting the E-Mission research project and constructing a sample use case involving smart thermostats. Modifying E-Mission services to run in our UPC implementation required minimal software changes, one requiring as few as 6 lines of meaningful changes. Additionally, experiments run using a Kubernetes cluster in Google Cloud showed that spawning each user's specific services may introduce a significant overhead, but if services can be effectively predicted, this increased cost is manageable.

Contents

Acknowledgments

I would like to thank everyone who assisted me in the process of completing my thesis and for giving me the opportunity to work on the E-Mission project. First and foremost, I would like to thank K. Shankari for her mentorship and patience when working with me. I am deeply thankful to her for giving me my first opportunity to engage in research as an undergraduate student, and I am especially grateful for her continued assistance after she received her PhD. I would also like to thank my advisor, Professor David Culler, for his involvement in this thesis. His guidance and input have been essential to this process, and he has given me a much better understanding of how to conduct quality research. I would also like to thank Professor Hellerstein for participating on my thesis committee and offering me valuable feedback on this work. Finally, I want to thank the other students who worked with me on completing portions of this project, especially John Sullivan for his guidance in understanding differential privacy. This thesis would not have been possible without the assistance of all of these people.

Chapter 1

Introduction

1.1 Consumer Data Collection

Within the past decade, technology companies have become synonymous with large scale data collection. This era, sometimes referred to as the Big Data Era [90], has allowed for the explosion of technologies which are only feasible on large input data sets, especially in the fields of Machine Learning and Artificial Intelligence. Simultaneously, the amount of participant data which can be requested has expanded. The biggest increases can be attributed to the proliferation of smartphones, which have increased in sales from 18 million in the United States in the year 2010 to 77.5 million in the year 2019 [62], and the expansion of the so-called Internet of Things (IoT), which are simple sensors that connect to the internet. IoT devices continue to expand in popularity. There were an estimated 27 billion IoT devices worldwide in 2017, and some experts speculate that there will be as many as 125 billion devices by 2030 [82].

Smartphones and IoT devices, in addition to increasing dramatically in number, also provide access to new types of data. Smartphones are especially useful for tracking location data, as they are equipped with a variety of sensors such as GPS and WiFi connectivity that can be used to acquire an accurate location estimate, and these devices are typically carried by people throughout the day. IoT devices often collect data for tasks that would not otherwise have available data. Examples include video streaming data that is captured by IoT door cameras and temperature sensor data from smart thermostats.

Technology companies often collect user data as part of their business model. Sometimes this data is used to offer services or increase the quality of products offered by the company, and other times this data is used to sell advertisements [87]. While these companies often describe data collection as a transaction in exchange for making products freely available, some researchers take issue with this business model. One proposal [73] argues that this approach is inherently unfair to consumers as they have no opportunity to ensure they are receiving adequate compensation for their data. Instead, the proposal argues that companies should have to buy data directly, rather than collecting it in exchange for a service.

Concerns about data collection are also based in part on the many data breaches that have impacted companies [81]. These breaches have prompted discussions about the need for increased consumer protection. This has resulted in some lawmakers placing increased rules and stipulations on how data is collected and the control companies have. For example, in 2018 the European Union passed the General Data Protection Regulation [1], which included, among other provisions, a person's right to erase their data, a right to request their data, and a right to explanation about how their data is being processed. Similarly, in 2020 California introduced the California Consumer Privacy Act [32], which gives a similar right to erasure. Additionally, in February Facebook [2] settled a lawsuit in Illinois over the collection of face recognition data without user consent [31].

However, these restrictions still fail to place users in control of data. Data collection is typically a process by which the company that produces an application or product gathers the data, processes it however it sees fit, and if a user wants that data deleted, requires the user to make a request to the company to do so. Often, this means the user must either allow their data to be utilized as the company chooses or abandon the service entirely. The procedure for requesting user location data from Google Timeline [3] illustrates this point: Consumers have control over whether or not any data is collected, but if they want the data they must download it in the Google Timeline format and must subject themselves to constant monitoring. Additionally, large multi-user data stores also present the opportunity for user data to be used by state agencies. For example, in 2018, California police officers used the tools to contact possible family members on GEDmatch [4], an open source genealogy website, to acquire a familial DNA match on a suspect in an old murder case [24]. More recently, in November 2019, a Florida judge granted police officers a warrant for the full GEDmatch data base [49], which some experts fear will lead to warrants for the data sets held by the biggest companies in the industry, Ancestry.com [5] and 23andMe [6]. As a result, despite efforts to increase consumer protection, users cannot properly dictate how their data will be used and by whom.

1.2 Differential Privacy

In response to the prominence of large scale data collection, many researchers are concerned with the amount of information exposed when using this data. One approach which has found popularity in the privacy community is Differential Privacy, which was first proposed in [38]. In this work, we focus on ϵ-differential privacy. In this definition of differential privacy, researchers are concerned with a database D and any database that differs in at most one row, D$'$, also called a neighboring database. ϵ-differential privacy is then concerned with the ability to distinguish between D and D$'$, which formally requires that for any algorithm A with positive parameter ϵ, for any subset S in the space off all possible outputs of A:

$$Pr[A(D) \in S] \leq e^{\epsilon} Pr[A(D') \in S]$$

Informally, this states that the relative information learned is bounded by specifying a value for ϵ. Meeting this definition then requires constructing a differentially private process. This is accomplished by adding statistical noise to the output published by a query to a given data set, and commonly Laplacian noise [40]. There are multiple options about where this noise can be added, most commonly referred to as local differential privacy when individual users apply noise on their own data and as global differential privacy when noise is added once to the whole database [36]. In this work, we focus on global differential privacy because local differential privacy typically requires more noise and is therefore less accurate.

ϵ-differential privacy additionally has the property that for n separate ϵ-private data releases from the same database, the overall information revealed is at worst $n\epsilon$-differentially private. This allows databases to participate in multiple online data releases through the use of a privacy budget [85], in which an initial ϵ is set for the database budget and each data release deducts from this budget its own ϵ'. Once this privacy budget is exhausted the database can no longer participate in data release.

One challenge for online database queries is that specifying results in terms of ϵ may prove challenging for data scientists who need a certain level of accuracy from their results. In response to this need, Sullivan [80] introduces a set of queries that allow ϵ to be specified as a function of accuracy parameters. In this work, we utilize the Acceptable Error query, which states that for count queries, publishing a final result which is within o from the true value with confidence $1 - \alpha$ makes

$$\epsilon = \frac{-ln(\alpha)}{o}$$

and the noise added to the result is $Lap(\frac{1}{\epsilon})$

A count query is a query where each row of the database can only have values 0 or 1 indicating whether or not a row is present. To generalize the Acceptable Error queries to other types of queries additionally requires considering the global sensitivity, Δf, which is defined as:

$$\Delta f = max\|f(D) - f(D')\|_1$$

,

Informally, this definition states that global sensitivity is the largest distance that results from applying f to any two neighboring databases as a function of the l_1 norm. Count queries are commonly used because $\Delta f = 1$. In this work we consider sum queries, which, despite having a larger Δf value, the literature suggests are applicable to differential privacy as the impact of Δf decreases as the number of participants increases [39]. Applying an Acceptable Error query to a sum query instead requires that

$$\epsilon = \frac{-ln(\alpha) \cdot \Delta f}{o}$$

and the noise added to the result is $Lap(\frac{\Delta f}{\epsilon})$.

1.3 What are User Private Clouds?

User Private Clouds (UPC) are a cloud-based architecture that seeks to provide users with control of their data in all stages of its life cycle. UPC seek to give users control over their raw data, how their data is processed, and to whom they supply their data. UPC utilizes a trust model where users are in charge of every meaningful decision about who and what can access their data. While perfect data privacy is not possible, UPC are designed to integrate global differential privacy, so users can still engage in meaningful large scale applications without sacrificing this control. In addition, all components are intended to be completely open source, so that any user with the means of inspecting how their data is being used can do so with complete confidence. It is our hope that the introduction of this trust model can empower users to be protective of their personal data while still participating in important research efforts. The UPC architecture was originally developed as a component of K. Shankari's open-source E-Mission [7] project. An initial version of UPC was introduced as part of John Sullivan's master's thesis on differentially private queries [80]. This report provides a full description of UPC and offers a sample implementation.

To illustrate the benefits of adopting a UPC architecture, consider the following timely example. Imagine an institution wanted to track information about the spread of a virus, for example COVID-19, and users wanted up-to-date information about where the virus has spread and if they are at risk. To accomplish this, one could utilize smartphone location data to institute a method of contact tracing. People who are diagnosed with the virus would disclose the locations they had visited, and other individuals who had been to those locations in the approximate time frame would receive a notification that they may have come into contact with someone with the virus. However, simply building this application with standard data collecting techniques would present some concerns, namely that users disclosing their data would not know exactly how it would be processed and might be concerned that their data would be sold. Similarly, fearing backlash, companies who collected the data might be hesitant to give this information to a third party without express consent from their customers.

A UPC solution aims to avoid these concerns by building such an application in a privacy-preserving manner. UPC are not the only platform design that strives to accomplish this. SafeTrace [44] aims to create a ”privacy preserving database-as-a-service” using a privacy preserving database and selected querying operations. It similarly allows for privacy preserving computation by running inside Intel SGX [25]. Where a UPC architecture differs is that it aims to be a general purpose architecture, whereas many solutions have been domain specific. In addition, UPC maintain a privacy-preserving data storage layer for each user, but then allow the user to manually select which services to approve.

Chapter 2

Related Work

In this section we detail works that share architecturally relevant similarities to User Private Clouds. We discuss how they have influenced the UPC design and how they should influence eventual UPC deployments.

2.1 Platform Services

The design points from which we initially drew inspiration were the commonly available Platform as a Service (PaaS) [50] options, such as Heroku [48], Google App Engine [27], and Amazon Elastic Beanstalk [29]. The most significant similarity between a PaaS and UPC is that both allow an individual to develop their own software stack through a combination of provided and uploaded services. However, in the PaaS setting, the user is generally seeking to deploy their own cloud application, so their design is oriented towards providing developers heavy redundancy. Instead, we were interested in providing each user with their own software stack directly, so users run independent services. PaaS also typically provides constraints on the types of software that are directly supported. Usually, this is a reflection of the architectural decisions. However, software constraints are also import in UPC for addressing privacy concerns. Finally, typical PaaS interactions place few restrictions on how developers collect user data. In contrast, the UPC architecture aims to design all applications to operate without having users sacrifice ownership of their data.

2.2 Cloud Application Services

Common cloud applications, or more generally Software as a Service (SaaS) [50], serve as a useful comparison because they are the most classic example of deploying computation in the cloud. SaaS is similar to the UPC model because both work to make services directly available to users. One big difference, however, is the role of users in the SaaS model. In the SaaS model users exist primarily through an account-based model. The SaaS provider tends to have full control of the user data it collects, and it manages user storage in aggregate databases of all related users. In contrast, UPC aim to give users greater data isolation and prevent SaaS providers from having direct access to raw data without explicit permission. SaaS is also similar because many of the technologies utilized by SaaS deployments often prove useful in UPC as well. One common technique for deploying SaaS is through the use of Linux containers, typically through docker images [43] and Kubernetes [70]. Although our UPC implementation uses these same technologies, users are associated with specific services rather than a broad set of services that are shared across all users. As a result, UPC cannot benefit from optimizations which function under the assumption that all users share resources. Finally, microservices [45], a programming paradigm for the development of SaaS applications which decomposes applications into smaller components for independent development and reuse, is applicable to UPC as well.

2.3 If This Then That

If This Then That (IFTTT) [52] is a service that aims at connecting user devices to enable complex personal applications. IFTTT has developers produce services, which fulfill some task on a device or software platform. Then, users can connect these services in a dependent sequence to allow for complex relationships, such as updating a spreadsheet anytime a user makes a request to a home personal assistant. UPC have similar requirements, but a major difference is in data processing. IFTTT services often operate by requiring a user to give complete data access to the service in question. In contrast, UPC seek to give users more fine-grained access control and ensure that data remains in user control except in uniquely necessary circumstances. IFTTT also plays no role in data collection. Although many IFTTT services are aimed at IoT devices, these services require either copying the data or requesting it from the provider. UPC seek to avoid this dynamic by serving as the sole target for data collection.

2.4 Password Managers

UPC also share similarities with password managers, such as Lastpass [42], Dashlane [66], Roboform [71], and ZohoVault [67]. Most password managers construct a password vault from which a master password can be used to extract a user's other passwords. UPC share this challenge of managing many keys through a master device key, but also need to hold additional secrets, such as any services a device trusts to run. Some password managers also need to facilitate external access to data. For example, ZohoVault is popular in a business context because it allows people to delegate a subset of passwords and permissions to their co-workers. UPC similarly need to delegate a subset of their data to their trusted services. One point of complexity in password managers also becomes synchronizing multiple devices, which similarly causes a split in the storage of password vaults across providers, with some relying on local storage and others relying on cloud storage. Cloud storage inherently makes synchronizing across multiple devices easier, but some users would prefer that all their passwords remain on their own device. UPC face a similar challenge because they are an architecture designed for users to connect all of their IoT devices, as well as their smartphone. While a person may want access to all of their passwords from all of their machines, an IoT device might not be trusted to decide which services can operate on the person's smartphone data. UPC are different from password managers because password managers are concerned only with account access and providing password access, but UPC aim to provide an architecture for password storage and subsequent data access. However, many of the engineering challenges associated with constructing password managers will apply to the UPC setting, especially when considering key sharing. One study [68] showed that many password managers struggle with scrubbing device secrets from derived passwords, a challenge UPC will share. Indeed, properly scrubbing secrets may be of greater concern with UPC because some UPC do secret processing in the cloud instead of just on user-owned devices.

2.5 Personal Data Stores

Personal data stores, such as Solid [88], MyDex [59], Hub of All Things [64], and OpenPDS [57] aim to be a single source for data storage. However, they differ from UPC because they are focused on enforcing access delegation among specified participants, while UPC restrict access to only specifically select code sections. OpenPDS attempts to give users greater privacy by defining an associated privacy cost lost by transferring metadata. However, OpenPDS appears to be an inactive project, and UPC opt for a simpler model of measuring privacy loss by relying directly on differential privacy for any global aggregation and keeping processed data within a user's control. Finally, personal data stores and UPC share issues with compatibility, an understandable challenge as most applications are not adapted to interact with them. Solid, a set of conventions and tools for designing social media applications, aims to do this by offering open source tools and allows for reusing existing data. UPC strive to ease this transition by instead relying on common cloud technologies, microservices, and API designs aimed at minimizing the need to modify software.

2.6 Personal Clouds

Personal clouds, such as Cozy Cloud [35], Freedom Box [69], NextCloud [60], and MyCloud [58], are systems which aim both to allow users to control all user data and to restrict the applications that can operate on data. Ideologically, they are the most similar to UPC. NextCloud, a platform for open source document sharing, has support for encrypted storage, an essential component of UPC. Many personal cloud solutions rely on achieving security primarily through hardware ownership. Freedom Box provides a software stack for users to download, but requires users to have their own server to be effective. MyCloud is similarly a hardware solution that one can purchase to act as an intermediate server. UPC do not explicitly require a hardware addition, but we discuss the benefits of adding one in section 7.4. However, although personal clouds allow data sharing, they do not provide any additional privacy to the user's shared data. UPC apply differential privacy to allow users to participate more privately in applications with large numbers of users.

2.7 Serverless Model

The serverless model refers to a model of application design in which developers no longer manage the servers responsible for delivering an application. Instead, application providers describe the computation and data storage necessary to perform an application, and the cloud provider manages the actual servers. There are many serverless model providers, but the most popular is the Lambda Model, offered by AWS Lambda [8] and given open source support outside of Amazon by the OpenLambda project [47]. The biggest similarity between UPC and the serverless model is that applications are also decomposed into small functional units and a known datastore. However, the functional units provided in the Lambda Model are required to be stateless and are shared across users. UPC has support for stateful services by providing each user with access to specific cloud instances, rather than a pool of functionality shared across all users. Additionally, UPC and the serverless model share an initialization bottleneck, in which spawning a service introduces a large initial overhead. AWS Lambda addresses this by reusing the same functional units across many users. We believe that UPC providers can tolerate this performance issue if they allocate common services preemptively based on expected demand. Other serverless models have also been developed more recently with more expressive behavior. For example, Cloudburst [79] extends the serverless model to have support for stateful Python applications. However, this stateful serverless model still differs from UPC because Cloudburst has no expectations of maintaining user data privacy while UPC expect user data to remain in user control even when being processed by an external service.

Chapter 3

UPC Architecture

3.1 General Overview

The UPC architecture consists of 3 primary layers, depicted in Figure 3.1. The storage layer holds all user data, both processed and raw. This information is stored encrypted at rest and can be decrypted using keys held only by the particular user to whom the data belongs. Next, the compute layer is responsible for all interactions with the storage layer and launching any services to process or stream the data to and from the storage layer. Finally, the user device layer consists of all user device endpoints, such as IoT devices and smartphones. These devices are responsible for data upload and selecting which services to run. All external applications must run through user devices. Every service occurs by having a user device contact the compute layer.

One of our primary concerns is usability, so our UPC architecture design is intended to reduce the burden on users of managing their own data. For this reason, the storage layer is intended to be a large horizontal layer, but with users controlling their own data. Similarly, the compute layer is intended to be employed as a cloud application, in which each user's service choices can be spawned on demand. We discuss the core components necessary to facilitate these services in section 3.3. Our decision to use cloud-based services is not absolutely necessary. We contrast the impact of alternative choices in section 7.4, some of which can offer increased security guarantees, but at the expense of individual upkeep or complexity. Finally, despite all interactions stemming from a user device, we believe that with a small device code base and a manifest of permissions which can be updated only with explicit user interaction, explored in section 3.4, fully automated applications can still function, but with the limitation that a user must approve of them explicitly.

3.2 Storage Layer

All user data is stored in a horizontal storage layer, as shown in Figure 3.2. Storage is partitioned across users, so that each user's data is isolated from all other users. Additionally,

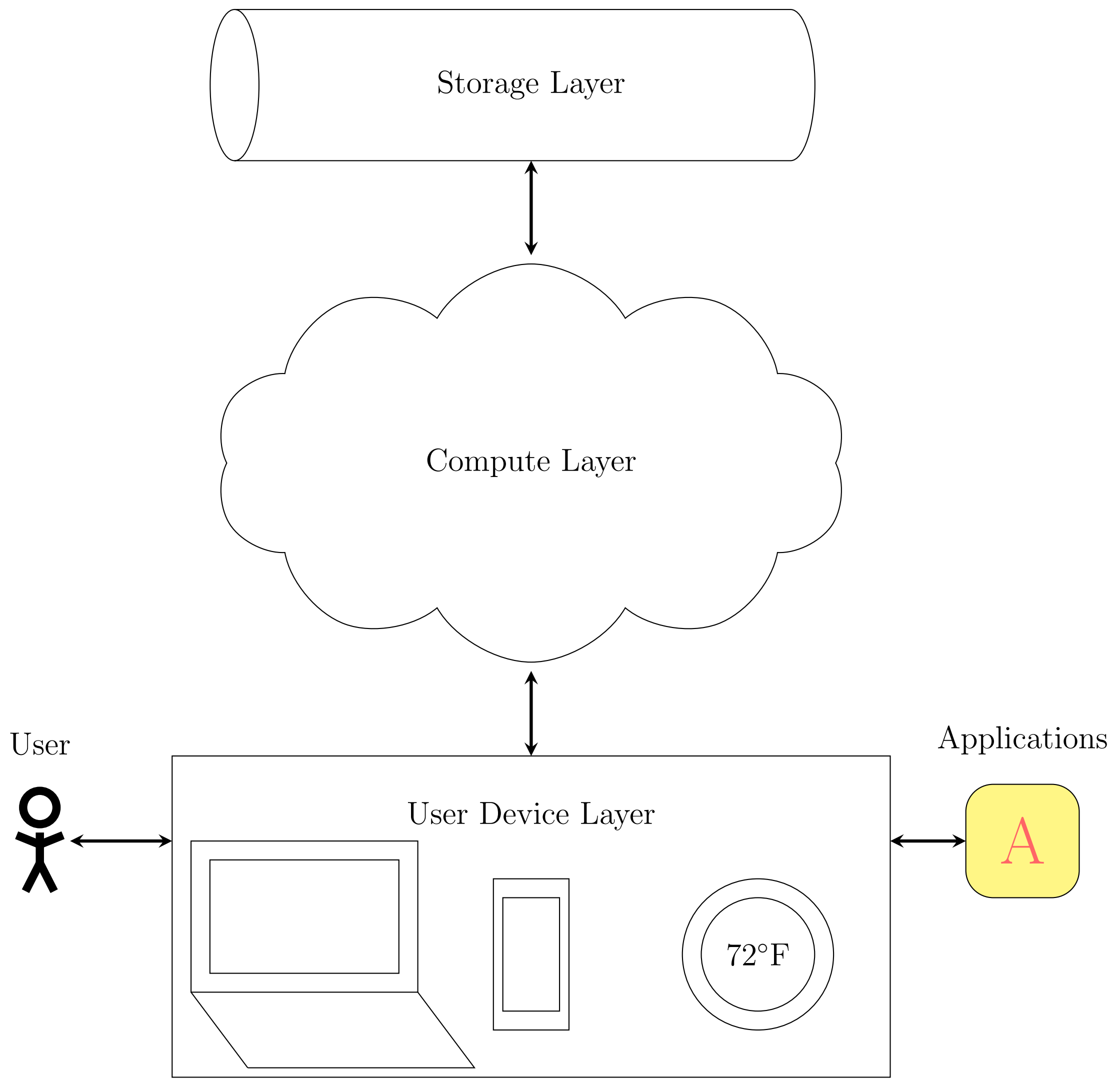

Figure 3.1: A simple depiction of the architectural components of UPC. The storage layer holds all data collected on behalf of the user. The compute layer is a cloud-based layer used to run services to manipulate data from the storage layer. The user device layer consists of all user devices that can collect data and decide which services are allowed to operate, such as laptops, smartphones, or IoT devices such as a smart thermostat. To interact with a UPC architecture, all users or applications must first contact a user device, which can interact with the compute layer to launch services, and these services can then interact with the storage layer to get data upon which to operate.

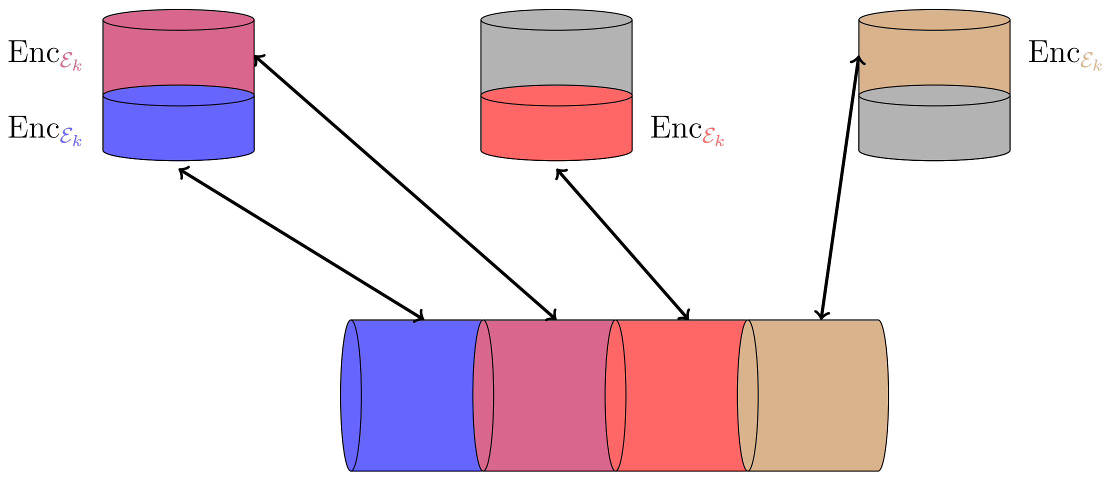
$\mathrm{Enc}_{\mathcal{E}_k}$
$\mathrm{Enc}_{\mathcal{E}_k}$
$\mathrm{Enc}_{\mathcal{E}_k}$
$\mathrm{Enc}_{\mathcal{E}_k}$

Figure 3.2: The storage layer is a logically horizontal layer of user data. The compute layer maintains a mapping of users to storage locations, each of which could be private or shared. However, all data is encrypted and each user's data has a unique key, requiring it to be isolated from other users.

each user must be the only individual that can directly access their data, allowing access by other individuals to only occur through a process of explicitly granting temporary access, explained in section 3.4. Meeting this requirement necessitates that all data is stored encrypted at rest and each user participating should have a unique set of keys to encrypt their data.

Implementing the storage layer allows for flexibility in how users decide to have their information stored. A user could provide their own storage, use a cloud storage provider, or use storage held by application developers. To allow compatibility across these implementations it is necessary to be able to locate a user's storage provider. To accomplish this, the compute layer holds a routing table for translating user identifying information to their respective storage locations. We assume each user's storage is a block storage device, and in practice we also expect it to be attached to a relational database running in the compute layer.

3.3 Compute Layer

The compute layer fulfills user needs for converting raw data into processed data and ultimately transferring it to the storage layer. Its purpose is to decompose traditional applications so that users control all intermediate forms of data. This means that applications now consist of different services, which function similar to microservices, in that applications are now decomposed according to specific actions that can be more easily replaced and shared across applications. To accomplish this goal the compute layer consists of three main com-

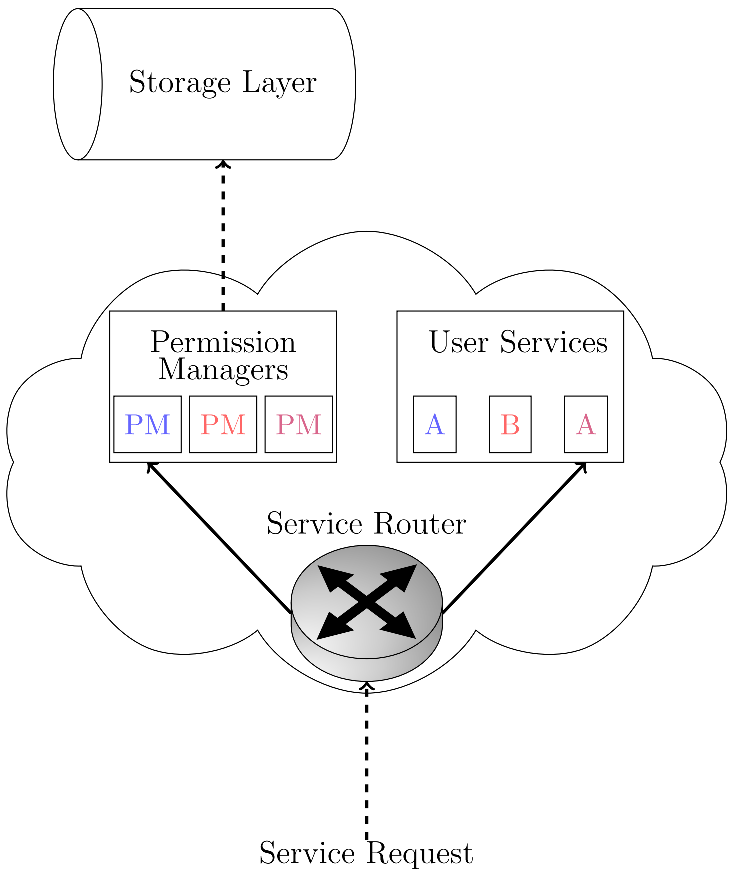

Figure 3.3: A diagram of the compute layer without the user device interacting with its services. The service router fields all service requests from user devices and spawns both the user requested services and the PMs. Each PM holds the keys necessary to decrypt data from the storage layer, so every time a user service is spawned, a corresponding PM must be spawned as well.

ponents. The service router is responsible for functioning as a published endpoint similar to many cloud based applications. It is also responsible for launching service instances, which are the actual computational units that either process or transfer data. Finally a permission manager is a special service for retrieving data from the storage layer. A diagram of the compute layer and its components is shown in Figure 3.3.

Service Router

The service router is the endpoint at which a user device initiates communication. While this component may be replicated using standard cloud application practices, the service router is a static component for user connections which should be accessible through outside means, such as name translation. When a user connects to the service router it indicates which services it needs to access. The service router then enables the fulfillment of these requests by launching service instances which correspond to the desired service. Then, the service router fulfills its purpose by supplying the user device with an address at which to contact their requested service.

Additionally, the service router is the only component of the compute layer which persists regardless of any explicit user action. Thus, the service router is responsible for fulling any global needs of the compute layer, such as providing a routing mechanism between a particular user and their storage location or tearing down finished services. This also implies that all user requests must begin with an interaction with the service router, as all service addresses are ephemeral locations. Finally, although the service router is not trusted with user data or secrets, it is responsible for validating the contents of all service instances. This could potentially have verifying requirements depending on provider desired rules, but at a minimum this should verify that the code image presented by a service instance matches the instance the user requested and that the user receives the address of the correct service.

Service Instance

A service instance is the main computational unit of the UPC architecture. Service instances are intended to consist of a small code base and perform relatively simple applications. As a result, a normal application will likely consist of many service instances. Any information sharing across these instances is required to interact through the storage layer, so users control all data that is released. Unless user authored, all service instances must be open source to properly allow users to inspect how their data is to be processed and that all approved services abide by their requirements (this could, for example, require that a service does no external networking). One concern with this approach, is that while open source code presents the opportunity for a dedicated community to review submitted services, this inspection process cannot be promised to be done properly and cannot guarantee the elimination of bugs. We discuss approaches to try and reduce these concerns in Section 7.2. Additionally, further experience may reveal that an alternative process, such as providing formal proof of service properties, may be better indications of code functionality, at which time it may be appropriate to amend this requirement. Nonetheless, we believe that until that can be properly demonstrated, transparency is necessary in all available services.

Permission Manager (PM)

The permission manager (PM) is a special service instance which is given the power to decrypt user data on behalf of the user by having the user stream it the necessary keys for decryption. It must be included with every service instance to allow access to a user's data and is key to the proper functioning of UPC. The PM is necessary because even if applications can access data from the storage layer, that data is encrypted, and only the user devices have the necessary keys. Functionally, this is equivalent to the user trusting each service it has approved with use of its keys for accessing data, but notable computing incidents, such as the DAO attack on Ethereum [78] have shown that validating open source code, even with an invested and interested community, is very difficult. As a result, we opt to introduce the PM as a requirement so that there is a smaller attack surface that must be verified in order to ensure user keys are protected.

Additionally, a PM could be greater empowered to allow for more rigorous control of data access. In section 3.4 we explore the primary mechanisms by which a user identifies services that it allows to process its data. In that mode we will consider access to only particular types of data and use that information to allow the user device to make the decision as to whether or not to launch the service. However, the PM could also be configured with more rigid data access restrictions. This could mean that data access could be further restricted to dynamic decisions about what data is requested. For example, a user might authorize a service to use location data, but only wants it to access data within the last month. The user device would then launch the service with the ability to request location data, but does not need to know the exact data that will be requested because the PM can restrict access to the last month.

3.4 User Device Layer

The user device layer consists of all user devices that can supply or request data. Ideally, this consists of a wide variety of devices such as smartphones, IoT devices, and personal computers. One of the goals of the user device layer is to limit device computation. However, one unavoidable increased cost is data transfer. This increased cost is the result of all applications needing permission to launch from a user device and also because all data that is eventually given to an application must first be routed through a user device. We require that all data stream through a user device because there are no publicly addressable entities in the compute layer that an application could access, as contacting the storage layer directly would present an application with data it can't decrypt, and any user services in the cloud exist at ephemeral addresses. Instead, the applications on a user device can be used to launch a set of services through the compute layer, as explained in section 4.1.

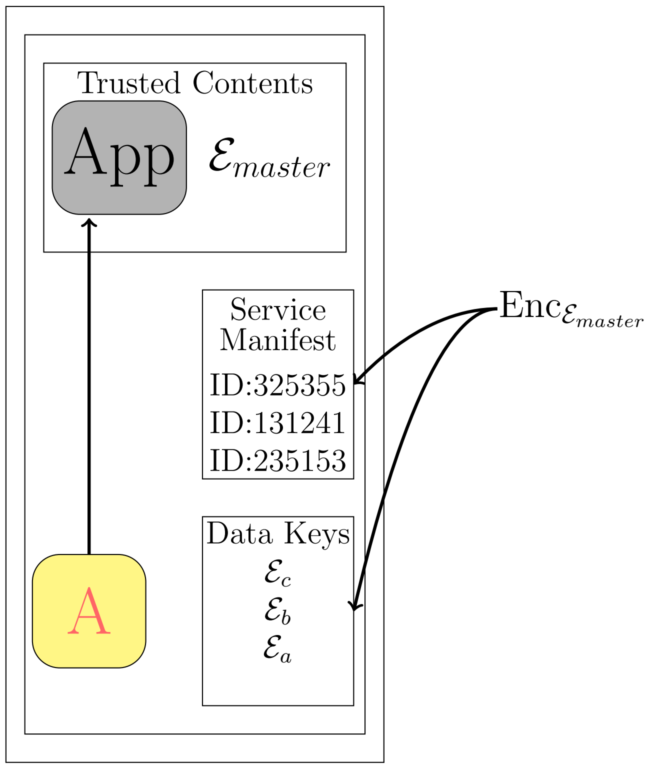

Figure 3.4: An example user device participating in a UPC architecture. The device is trusted to store a master encryption key and have privileged code for using the key and contacting the compute layer. The device also contains a service manifest, which lists and describes all user approved services and a set of encryption keys used for storing data in the storage layer. Both the service manifest and data keys are encrypted by the master key so only the trusted code segment can modify them.

In addition to sending and receiving data, the user device also tracks which services should be launched. The service manifest, which can be held separately by each device or jointly across all user devices, holds the set of services a user approves and what data they are allowed to access. In this work, we consider a simple data permission system, in which a device has or lacks access to a particular type of data, such as GPS or accelerometer output, from a particular device. Additionally, user devices also contain a set of keys for editing the service manifest and encrypting data. An example of the contents of a user device is shown in figure 3.4

Although in our examples we consider a user device layer as a simple broad grouping of user devices, a more complex device layer configuration could prove to be useful. For example, if a user's home thermostat is trusted significantly less than their smartphone, it may prove fruitful to give the thermostat access only to the keys to edit the data it can provide and not to give it the means to modify the service manifest. Additionally, if a user has a device through which it is faster or cheaper to stream data, it may be useful to equip that devices with more keys and configure applications to initiate service requests through the device. We explore how alternate deployments, especially changing how the PM is deployed, could impact the application process in section 7.4.

Service Manifest

The service manifest holds the information necessary for a user device to approve a particular service. This includes at least an identifier for the contents of the approved service such as a hash, a list of which applications can request a particular service, and which data streams a service has permission to access. The service manifest should be able to edit only with explicit user action and should not be able to modify automatically. Thus, if one were to add enhanced security, such as through 2-factor authentication, it would be ideally placed on the service manifest.

The mechanism for distributing a service manifest across user devices may vary. The simplest approach would be to give each device a unique manifest and allow it to approve service requests to data only if the data originated from that device. However, this would be limiting as it would prevent services from simultaneously accessing data from multiple devices. To alleviate this concern, it may be desirable to share a single service manifest across devices and store it in the storage layer, with downloading the manifest available as a service. Alternatively, some devices could be given manifests which are super sets of other manifests, such as empowering a smartphone to approve operations on an IoT thermostat, but not let the thermostat approve services not directly operating on its data.

User Keys

In order to address the constraints of ensuring the storage layer data is accessible only by the user, each user device is responsible for holding multiple keys. The first key is a device master key, which is used to edit the contents of the service manifest as this manifest should be stored encrypted on the device using the master key. Then, this key should be used to generate a set of data keys which are used to store a user's data in the storage layer. For our implementation, we opted to use a separate key for each data stream, but this is by no means a requirement. Then, when a service needs access to a set of data type on each device, the PM is sent the necessary set of keys as the service is launched, so it can facilitate the requests. If, as previously mentioned, the service manifest requires synchronization across devices, then it is necessary to share keys across user devices or derive keys from multiple device keys.

3.5 Threat Model

In this work we consider a *semi-honest* threat model [46]. We trust the cloud service provider to correctly run any necessary software, but the provider or its administrators may attempt to read anything stored in non-volatile memory. We do not explore any issues associated with denial of service attacks. Furthermore, we partition our architecture into various trusted components. We fully trust any user devices to properly execute a small code segment responsible for interacting with the cloud provider and to check the necessary permissions in the process. Additionally, we fully trust the PM. Most other components we trust to fulfill their assigned task, but do not offer them any secrets. For example, we trust the service router to accurately launch the services a user requests, as well as to route users to the proper service when possible. Similarly, we trust any user-approved service to process data as advertised and not to steal or copy a user's data, which can be partially ensured due to the open source nature of the services. These assumptions may not always be realized in practice, but because these entities have no access to user secrets the resultant loss of private data is limited.

One task in meeting our threat model that may take some significant engineering consideration is ensuring a user's keys never get swapped to disk through process swapping. In our sample implementation, we have not taken extensive measures to prevent this, and anyone using a UPC architecture should be cautiously aware of the need to wipe any secrets from memory. Rather than focus on handling this difficult engineering challenge, we leave it as an implementation consideration and note that alternative deployments in section 7.4 may be a possible substitute.

Chapter 4

Using the Architecture

4.1 Launching a Service

When launching a service utilizing the UPC architecture we assume the following setup. First, the user carries at least one device that has already registered with a configuration, meaning it has location allocated for it in the storage layer and that the service router has this location in a routing table. Next, any application that we are considering has an appropriate code segment on the user device from which it can contact the device, and it has been fitted with the appropriate permissions to operate. This includes any user requested services, which we will also assume are routed through some external application. Finally, we assume that a user has already added any service they wish to run in the service manifest and has created the appropriate keys, which are accessible only from a privileged process that the application wishing to run a service must contact with a request to launch a service or obtain data. The steps to launch any service are:

1. An external application first initiates the launch of the service. This can either be a user who manually requests a service through another application or a background effect of an application or, alternatively, an application automatically running an application. In either case, this application presents a request on a user device to the main UPC protected application.

2. The privileged device process checks the requested service against its service manifest using the device master key.

3. If the service is verified, the user device contacts the service requesting a service instance.

4. The service router launches a new service instance and a corresponding PM.

5. The service router returns the address to contact the new service instance, the address of the PM, and the address of the storage layer from its routing information.

6. The user device then contacts the PM. It sends the private key(s) necessary to encrypt/decrypt data to the PM and the address of the user's storage layer slice.
7. The PM verifies that the key(s) match the user via authenticated decryption and responds to the user.
8. The user then contacts the service instance with the address of the PM and any information necessary to launch the service.
9. The service instance makes requests for data to the PM and fulfills its purpose. If it needs to store any data, it also sends it to the PM. If there are any additional permission checking steps implemented, then these are fulfilled by the PM.
10. The service instance responds to the user device with an error code and any data the device requested. The user device then forwards this information to the user application that initially requested the service.
11. The service instance contacts the service router indicating that it can be decommissioned, and the service instance decommissions both the service instance and the PM.

Notably, this procedure does not include any optimizations or changes that arise from an alternative deployment. For example, if a user had multiple outstanding service requests, the service router could try to spawn a single PM instance for usage across all services, or a service router could preemptively spawn services that are heavily requested. Our implementations will rely on the basic implementation procedure, and we will not attempt to offer any quantitative insight to the advantages presented by modifying it.

4.2 Aggregation

Applications can be constructed simply by connecting services and having each service undergo the process in section 4.1. However, even if applications use the service model from the UPC architecture, this may not be sufficient because many applications still need to access to user data directly. If this data is eventually streamed directly to the application provider, then the user loses control of that data and is again relegated to trusting the application provider to engage in appropriate use of their data.

To address this concern, we consider applications that engage in aggregation, which is the process of combining data from multiple users to fulfill some joint goal across the data. For our purposes, aggregation refers either to the combination of a small number of users' individual data to achieve some collective goal, which we call local aggregation, or the combination of many users' data with the goal of achieving a holistic representation across many users, which we refer to as global aggregation.

Local Aggregation

Local aggregation is used as a means of combining data from a small set of users to achieve a collective goal. This form of sharing arises naturally in many applications, such as a calendar event making a booking on two users' individual calendars. Additionally, another motivating factor is a device shared by multiple people which makes a single decision based upon each person's data. We explore this case in section 5.3 in the context of a thermostat trying to minimize its power usage.

A defining characteristic of the local aggregation process is that, because of the small number of users, any data that is transferred cannot be obscured through a process such as differential privacy. This means that either the application is trusted to access the raw data directly or an alternative data oblivious process, such as secure multi-party computation [92], must be utilized. For this work, we assume that local aggregation requires directly sharing a small amount of user data. This is accomplished by first executing a service instance on behalf of the local aggregation to process the data and then sending the output to an aggregation service accessible by all users. We assume that to execute this aggregation service, any shared device will have access to its own UPC instance, including storage.

With local aggregation many concerns could arise as a result of ownership and domain considerations. For example, questions may arise as to who should have access to a device and who should be empowered to select and inspect the services that can be performed on it. However, this work makes no attempt to address these issues.

Global Aggregation

Global aggregation, in contrast to local aggregation, is intended for applications that require pooling data from a large number of individuals. In this setting it is not expected that individual data points are particularly important, as inference is drawn from the combination of data points. This allows user to obscure this data in order to provide better data anonymity. For this work, we require that every global aggregation be produced while adhering to differential privacy, whose details, such as the privacy budget, and the amount of privacy leakage expended by each data point, should be tracked directly in the storage layer. A reader unfamiliar with differential privacy should refer to [61] as a simple introduction and then read section 1.2.

UPC are designed to accommodate global differential privacy, where the application that collects data across many users applies the actual noise. UPC are able to support global differential privacy because every component involved in the query is directly trusted and approved by the user.

One additional possibility is to introduce hierarchical aggregation, where a global aggregation is performed over many local aggregations. Privacy budget tracking is now more complicated because the local aggregation unit may participate in multiple global aggregations. If this data relies on data from multiple users, then those original users need to track the privacy loss associated with the local aggregator using their data. Additional coordination is also needed to ensure each local aggregator obeys the privacy budget of each of its users, requiring additional online steps. Finally, to ensure users have proper autonomy over the usage of their data, a local aggregator should have a method of approving services or global applications that requires the consent of all users or otherwise allows the exclusion of data from users who do not approve of the application. An example of how this could be achieved would be to implement a local aggregator as a separate UPC user and make the master key able to generate only with the participation of all users, such as through Shamir Secret Sharing [76]. This example would be problematic for revoking service approval, however, so an implementation of this approach would require a more complete protocol.

4.3 Transitioning to UPC

While sections 4.1 and 4.2 illustrate design principles that may be useful for creating a new application, they offer little support for application developers who are seeking to transition an existing application to support a UPC setting. We consider transition existing software to be an essential feature of any new approach, so in this section we introduce the UPC Service API. Then, we explain the steps necessary to transition an existing application by comparing the steps necessary for users to upload data for services processing data, and finally for services that seek to analyze data over all users.

UPC Service API

The UPC Service API is our attempt to shift the burden of application transition from the application or service developer to the UPC architecture developer. Consider an application that processes data by making calls to and from a shared database across all users. Our expectation, shown in Figure 4.1, is that such an application engages with a programming language API, rather than just the database. To naively transition this application into UPC would mean transitioning every single database call made by the application to a direct interaction with a PM and, in the process of doing so, evaluate any assumptions about when or how the data expressed by the database's API is presented.

Rather than asking application developers to make these transitions, we instead propose designing Service APIs that emulate the functionality offered by common database APIs. In this setting, each Service API contains function signatures that match those offered by the database APIs. However, these implementations will instead perform transactions in a UPC style, contacting the PM directly for any data querying or storage and presenting proper credentials. The PM itself should also have an accessible API, so users can have access to any functionality which is shared across all services, such as deducting from a user's privacy budget. Then, the PM will interact with an underlying database through a programming language API. This interaction is shown visually in Figure 4.2.

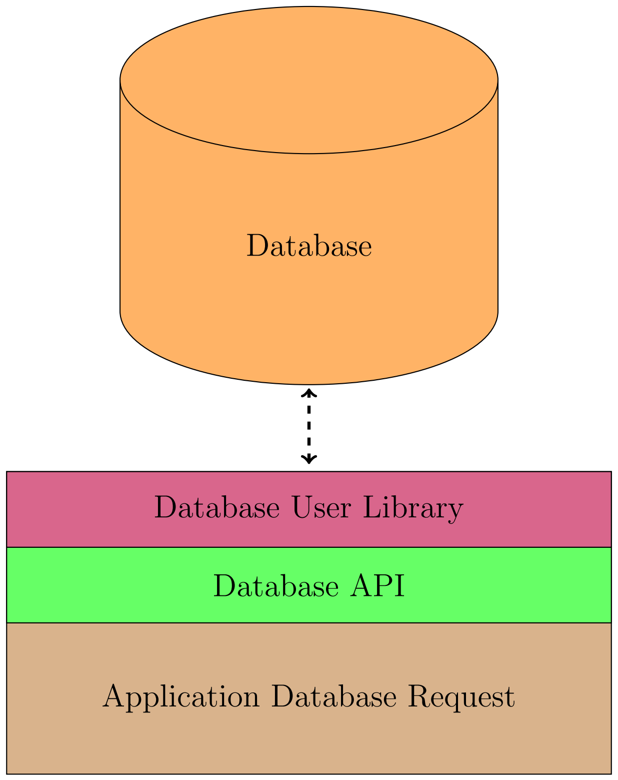

Figure 4.1: In a standard non-UPC application, the source code stores data by interacting with a database API implemented in the programming language used by the application. This API is fulfilled by a library which interacts with the actual database.

We believe there are three key advantages offered by designing the UPC Service API in this way. First, while UPC architectures are required to implement expressive database APIs, application developers need to make relatively few changes. We discuss the specific changes below, but in the simplest case a service only needs to make the UPC Service API aware of the PM to which it wishes to connect. This is not promised to be optimal, as for example any code which stores or retrieves values by searching for a per user value will still store such values when they are no longer needed, but the barrier of entry into UPC is still significantly lower than needed to reconfigure the entire application to understand it is connecting to UPC. Second, the UPC Service API offers a clear approach to connect services that assume different database technologies. Services that were written to operate on a different database can each operate on the same PM instance because their database specific APIs handle the translation, and any underlying database used by the PM is not exposed. This model is demonstrated in Figure 4.3. Third, the UPC Service API shifts the burden of updating database modules away from developers. Developers only need to modify their code if they want the newest features because the Service APIs made available are static for any versions

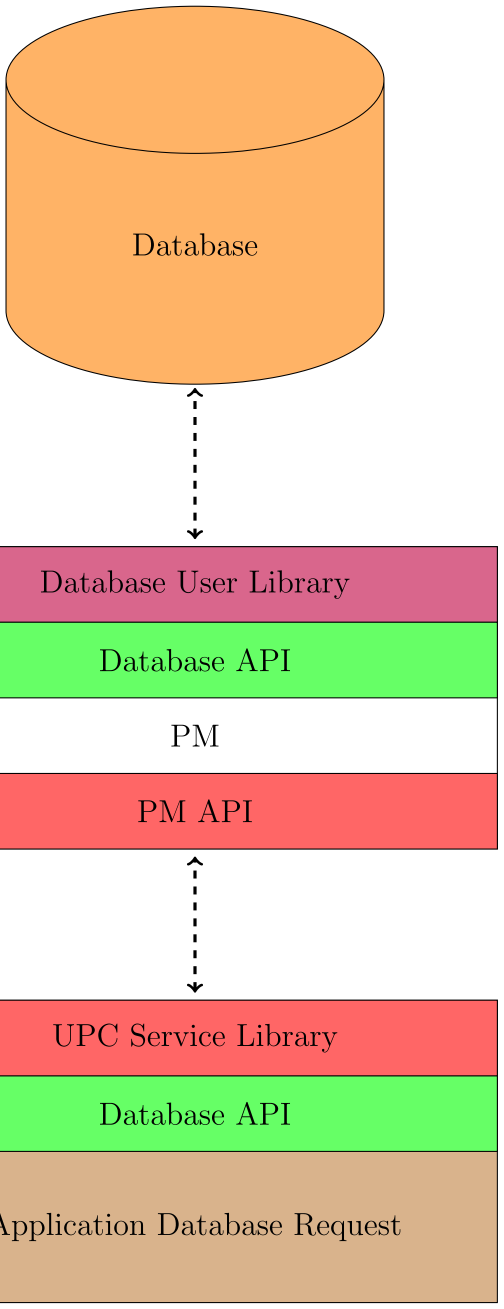

Figure 4.2: The UPC Service API allows for applications to see the same database API for data storage, but it replaces the underlying library with a library that transforms the request into one that engages the PM, *i.e.*, the UPC Service Library. The PM then takes that request and makes an actual database request to the storage layer.

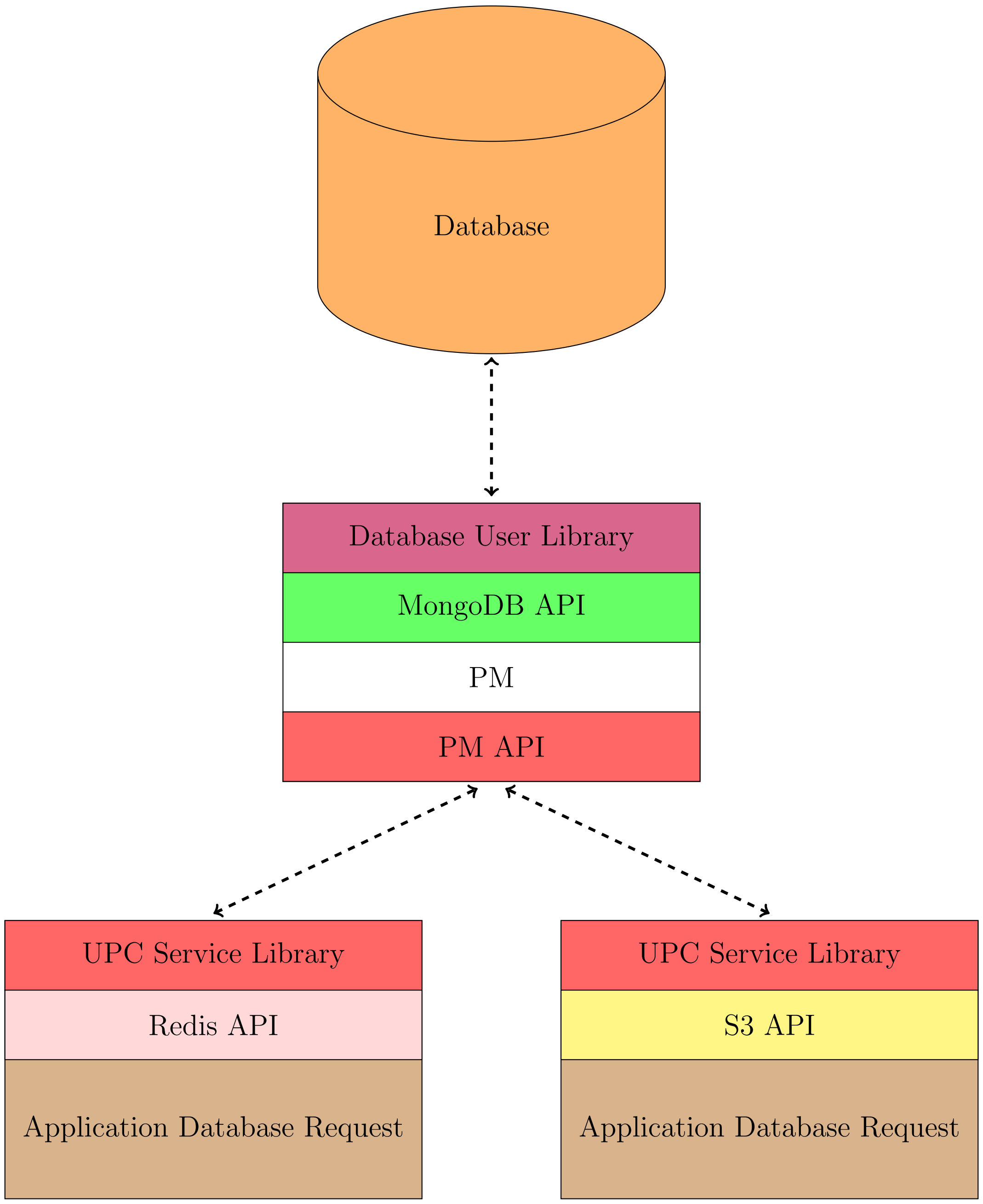

Figure 4.3: The UPC Service API allows multiple applications that expect different underlying databases to interact with the same PM without significant changes to their code base. A UPC Service Libary is implemented for a set of supported databases, each of which transform their request into a PM request, which may operate on one or neither of the databases with which the applications believe they are interacting. In our example, the PM is interacting with a MongoDB [83] instance, but the applications which they service believe they are interacting with Redis [9] and Amazon S3 [10]. This approach can also be leveraged to support multiple versions of the same database API.

that are supported. Instead, transitioning versions are the concerns of the UPC Service Libraries, which perform the actual transformation, and possibly the API offered by the PM. While the Service APIs should continue to offer front end implementations of the latest database versions, applications should only feel the burden to transition to these if they seek to express new functionality offered by the API change.

Finally, we consider the specific steps necessary to transition an application. We initially assume a monolithic structure and then attempt to demonstrate the minimum steps necessary to transition the application to UPC with a UPC Service API. For simplicity, we assume the application has three main parts: data upload, which is how users initially upload data to the database; data processing, which is any code written to transform user data from raw data into a more complex representation; and data analysis, which is any code that derives insight from looking at data across multiple users. These do not represent distinct application stages and may be scattered throughout an existing application, so identifying these components may be a necessary first step. Additionally, we do not discuss the code burden necessary to make requests to launch the PM.

Data Upload

Uploading user data in many monolithic applications likely involves transferring data to a centralized server that processes data storage. To add UPC support for data upload, this database upload needs to be directly exposed to the user application. This may be as simple as moving the database calls from the server into the user application and replacing them with the UPC Service API. However, the server may have initially processed the data before storing it, so this may also require adding a service to process raw data into the representation and location expected by the rest of the application.

Data Processing

The UPC Service API aims to require the fewest changes for services that process individual user data. Integrating data processing services should only require interacting with the UPC Service API and never the PM API. The three changes needed are:

1. Create a service entry point that can be run as a server. This can be imported from one of many server modules and only requires a common listening port.

2. Collect the address of the PM from the the service entry point and make it available to the UPC Service API.

3. Replace calls that initialize the database with calls to initialize the UPC Service API.

The UPC Service API should be expressive enough that all database call and table references can be expressed exactly in the API without any further modifications. This does not promise to offer an optimal implementation, which likely requires careful consideration of which features in the centralized database need to be kept in each user's database, but it should provide a low barrier for entry when adapting code.

Data Analysis

Data analysis requires decomposing the code section response for aggregating user data into two components: a service responsible for querying the necessary application and an aggregation step which computes the final differentially private result. The service creation should resemble the previous steps for data processing, but also require a call to the PM API to deduct from a user's privacy budget before obtaining the results of the query. The aggregation step, which will apply the appropriate noise to the overall data, can be run outside the UPC services, but as mentioned in Section 4.2 this code is still trusted to be open source and able to verify its identity.

The greatest challenge for application developers will probably be implementing differential privacy properly and enforcing budgets. While a future goal would be to have standard libraries available for application developers, some details will still rely on application developer decisions, such as selecting ϵ or calculating the global sensitivity for the set of databases in consideration.

Chapter 5

Implementation Examples

5.1 UPC Architecture Instance

To implement an example UPC architecture we focused on implementing the compute layer and opted not to implement device applications for the user device layer or a distributed storage layer. As a result, we have not implemented a connection directly with any user devices, although we do not consider this action essential because for many devices building an application to run on the device is standard practice. We execute our computer layer using docker images run inside of Kubernetes. Docker images are used to specify Linux containers consisting of the necessary environment and application code. Kubernetes is then used to orchestrate these containers across a cluster of machines, as well as specify information such as storage configuration and restrictions on network access. We primarily rely on the pod and service Kubernetes abstractions. The pod abstraction describes a set of related Linux containers which may need to communicate. The service abstraction indicates how a pod can be accessed from outside the cluster of machines hosting Kubernetes.

The service router is written in Python3 and is an extension of the open-source bottle server [72]. Services are deployed in Kubernetes using the NodePort service abstraction. This abstraction directly exposes a static port and IP address on the cluster which the service router uses to give each user a unique service. However, Kubernetes expects users to operate a small set of services and replicate the instances supporting that service across a cluster. While some stateful connections can be maintained, this model provides no mechanism to ensure each user connects to a unique container.

To circumvent the Kubernetes expectations, we create a unique service for each user. There is no mechanism for directly using the same service configuration file for multiple services, so we create temporary files using the Python3 standard library's Named Temporary File and create a unique temporary file for each user. Then, by changing the described name, selector, labels, and ports, each service will appear unique.

The PM is implemented as another set of docker containers run inside Kubernetes. One container runs the PM API, and the other manages a MongoDB instance. To keep data encrypted at rest, we mount the Linux encrypted file system eCryptfs [41] from inside the MongoDB container. This requires giving these containers elevated privileges and requires eCryptfs to register the necessary keys in the kernel key ring. We have experimented with this process and found that immediately clearing the key ring after launching the mount allows the container to continue to read and store encrypted data, but we have not been able to read that data outside the container. However, this still introduces an attack through which, with appropriate timing, a malicious operating system could extract a user's key, motivating our decision not to entrust the operating system to malicious log keys. We did not integrate persistent storage volumes for our sample implementation.

To implement the Service API, we constructed a set of Python3 classes that implemented a subset of the pymongo [11] 3.5 API necessary to handle the functionality used in E-Mission. This involved adding support for 5 operations: find, delete, update, insert, and replace. Our implementation queried the database lazily, only contacting the PM whenever data was expected by the program in the hopes of mirroring the performance impact of operations in pymongo. We also needed to add deducting from a user's privacy budget to the PM API.

5.2 E-Mission

E-Mission is an open source platform designed for aiding transportation researchers in gathering human mobility data. It the subject of K. Shankari's PhD thesis [77] and was originally the basis for UPC. E-Mission was designed in the central server model, in which researchers conducting a travel study would collect raw data generated by the participants' phones. Additionally, E-Mission provided inference runs on this centralized server.

To transition E-Mission to adapt to the UPC architecture, we extracted two services from the central server. The first is the intake pipeline, which takes raw data collected from smartphones and processes them into trips, denoting a perceived mode of transportation. To change this code, we only needed to expose the intake pipeline command as a server. We did this by extending the bottle server and replacing the existing tables referenced through pymongo with calls to our UPC Service API. Ignoring the code that we took directly from the existing e-mission server to make the intake pipeline available as a server, this meant that we only needed to change 6 lines of code to replace pymongo with the UPC Service API. We then needed to make about 50 lines of additional changes to instantiate our database tables properly, but this could easily be optimized out with a small refactoring to the UPC Service API implementation. It is worth nothing that this was not an optimal adaptation, as the intake pipeline still checks many values which are only logical in a shared database. While the performance impact of optimizing for a private database is interesting, for this evaluation we were only concerned with making the minimum necessary changes.

The second service we extracted was an analysis service for reporting user metrics. The metrics supported are a count of trips, the total distance traveled, and the total amount of time spent on trips for each mode of transportation. To adapt this service, we once again had to replace any database calls with the Service API and make the analysis service available as a standalone server. To enable the service to participate with a global aggregator, we also had to include a function call to subtract from the privacy budget.

To construct a global aggregator that gave differentially private results, we adapt the results given in the context of an early version of UPC in [80] as mentioned in section 1.2. This work notes that spatial temporal data is typically not applicable to traditional differential privacy because it violates the i.i.d assumption among rows in a database. However, to regain independence among rows, we define our database as consisting of one row per user and instead assume independence among users.

Our queries are all sum queries, so we select a Δ f as the difference between the maximum and minimum value possible for any mode of transportation. While all minimums are clearly 0, maximum values are not mathematically certain for our queries. We determined our Δ f values by making assumptions about speed and looking at data from the 2017 National Household Travel Survey (NHTS2017) [22]. These values are given in Table 5.1 under the assumption that the queries must be conducted in multiples of 24 hours. We do not consider them optimal, especially when a query asks about many days, but we believe they should be sufficient to ensure differential privacy.

Query Type	Δ f Car	Δ f Walking	Δ f Bicycle	Δ f Transit
Number of Trips	55 trips	55 trips	55 trips	55 trips
Distance Traveled	3840000 m	504000 m	648000 m	1440000 m
Time Traveled	86400 sec	86400 sec	86400 sec	86400 sec

Table 5.1: List of Δ f values used for query calculations per day. Trips are not well defined. So, we selected a number bigger than the most trips with any mode of transportation from NHTS2017, 48. For distance, we give a unique Δ f value for each mode of transportation supported.

5.3 IoT Thermostat

As a second example, we opted to model an IoT thermostat's application written for a UPC architecture. Our example of interest is setting the temperature for a thermostat shared by multiple people who occupy the same living quarters. The aim of the application is to determine when people are expected to be home so the thermostat knows when to set the temperature. We opt to meet these goals using a set of users with calendared work events, and we consider the last event on the calendar to be the end of their workday (and thus when they will go home). The application will consider all users' calendars and then decide when it needs to start adjusting the temperature and to what temperature it needs to adjust. Calendars were created in Python3 using a standard icalendar format and uploaded to the storage layer of a series of users. Then, a simple application to scan a user's calendar and to determine when to set the temperature was written in Python3 and deployed as a service using Kubernetes.

This example was selected as an example of a local aggregator. Our generated users all transfer their data directly to the device because they all trust each other and all seek to obtain the benefits of giving the IoT device access to their data. In addition, this example also illustrates how IoT devices may interact in a UPC setting. This serves as an example of a cross-device application, since the calendar information is derived from applications on a user's smartphone, whereas the temperature setting information is from the thermostat. While this example attempts to show how an IoT device could be integrated, we did not actually use a real thermostat nor did we modify the data sent to its storage layer to conform with the data output of any particular thermostat. We also omitted the final calculation of expected arrival for each user.

Chapter 6

Evaluation

6.1 Evaluation Setup

To evaluate our design, we conducted a series of experiments on a Kubernetes cluster hosted on the Google Cloud Kubernetes Engine [12]. Our cluster consisted of 4 n1-standard-1 machines, each with one virtual CPU and 3.75 GB of RAM in the us-west-1a zone. We ran all UPC experiments across all 4 machines. None of the containers that we tested were optimized for CPU and memory limitations, but instead were configured with the default settings. As a result, despite Google Cloud having support for up to 110 pods per node, we were able to launch only 27 on the whole cluster before available CPU was exhausted.

While our architecture without optimization requires dynamically spawning each container, to enable larger experiments in a more timely manner, we opted to spawn all containers once at the onset and only access the time necessary to perform the tasks of each container. We expected that for any service whose image is already on the host machine the actual time to support such a service lies between this reported time, which represents a best scenario of the architecture effectively predicting the demand for certain containers, and an increased latency which is proportional to the times given by the initialization micro benchmark. We were unable to run an experiment in which the machines repeatedly needed to load the image from a remote container registry because there is no mechanism to delete local image remnants on the cluster. However, on the few instances we witnessed the need to load the image from remote storage, it was between one and two minutes.

6.2 Micro Benchmarks

To analyze the cost of running UPC applications, we divided the steps involved in running UPC into a set of micro benchmarks. First, we considered the cost associated with dynamically launching a UPC service, which consists of spawning one pod and service. To determine initialization costs when the necessary images are present on the machine of interest, we measured the time from when the request was made to the service router until the pod of interest was considered active. We did not need to measure the time to spawn the service because NodePort type services are simple enough to avoid any significant delay. We sought to determine if image sizes or the presence of other inactive pods contributed to the delay. To do so, we examined three services: the calendar service, a 411.91 MB image; the pipeline service, a 1.22 GB image; and the PM service, which relies on two pods, one with a 411.07 MB image and the other with a 197.87 MB image. We also examined whether additional pods which rely on a 197.87 MB image impacted initialization cost, although we did not impose any workload on these pods. The experiment consisted of 20 UPC service spawns with 0, 10, and 20 addition pods in the cluster. We then report the mean and standard deviation of the last 19 requests in Table 6.1 to omit any cost associated with initially needing to remotely load an image onto the machine. Our results suggest initialization is consistently between 2 and 3 seconds, regardless of image size or additionally running pods. Of course, this neglects the additional differences in time to download larger images when the data is not present.

Another feature which may be necessary to measure, as opposed to a traditional Kubernetes approach, is the time to shut down any existing pods. As each user has a possibly unique software stack, it may be necessary to remove pods after use to accommodate the next user. While a provider can also expand their cluster, this may prove to be too wasteful in practice, especially since services with state need to be deleted after they finish processing data. We examined the time necessary to remove each of our three services when they were the only pod on the machine in Table 6.2. Surprisingly, the PM service is six times more expensive than shutting down a stand-alone pod. This is likely due to the presence of multiple pods in the service, but it is unclear why the delay was not proportional to the number of pods. Notably, none of these pods were doing meaningful work, and none held persistent state, so shutdown costs may be much higher in that situation.

Service Name	Other Cluster Pods	Mean Creation Time (s)	Standard Deviation (s)
PM	0	2.677	0.536
PM	10	2.765	0.529
PM	20	2.400	0.262
Calendar	0	2.568	0.499
Calendar	10	2.418	0.387
Calendar	20	2.290	0.050
Pipeline	0	2.457	0.377
Pipeline	10	2.311	0.049
Pipeline	20	2.321	0.098

Table 6.1: Time necessary to spawn each pod with possible other pods already running in the cluster. This data is averaged over 19 attempts, and the images used by the pods are already local to the machine.

Service Name	Mean Delete Time (s)	Standard Deviation (s)
PM	36.19	1.990
Calendar	6.715	2.966
Pipeline	6.036	3.350

Table 6.2: Time necessary to delete each pod with no other pods already running in the cluster. This data is averaged over 20 attempts, and no pod was sent any requests or data.

Our next set of micro benchmarks sought to analyze the additional cost introduced by utilizing the PM. Our first set of tests analyzed the time taken to upload and download large sets of data from a client machine outside the cloud to a PM inside the cloud. We display these results in Figures 6.1 and 6.2. Not surprisingly, upload and download times scale proportionally to the amount of data uploaded. Similarly, network bandwidth is clearly the limiting performance factor, most likely due to bandwidth constraints on the client machine. The increased time associated with increasing the number of concurrent PMs also seems to scale purely as a factor of the amount of data being uploaded or downloaded, so it seems unlikely PM allocation will be a significant performance hindrance. Our data also suggests that the client upload speed was the bottleneck, as uploading data took longer than downloading it.

Finally, we wanted to determine the expected cost produced by running applications as a set of services rather than as a single service. To do so, we opted to try to analyze the overhead introduced by communication cost and the cost associated with data transfer. To get a baseline of the cost introduced by using services, we analyzed our Calendar Service, which we configured to query for 1.1 KB of data with various size databases. We plot these results in Figure 6.3. We then compared this to an alternative service which made a request for all of the contents in the database shown in Figure 6.4. Our results suggest that for a small database, partitioning into multiple services has a low communication cost, but unsurprisingly since the UPC Service API relies on database requests, communication cost grows with the size of the database being queried. Larger data transfers also seem to be limited by network speeds. Examining the transfer times when running a single set of services, there seems to be no significant advantage to transferring the data within the cluster as opposed to outside the data center. Instead, it seems likely that increases transfer costs when increasing the set of services are instead primarily due to transferring larger amounts of the data to the host machine. When only two sets of services are running, each service can run on a separate machine, which likely explains why there is only a slight increase in transfer times. However, when switching from 2 to 4 sets of services each machine must have at least two services running, increasing the network demands on each machine and thus the overall transfer time. It also appears that multiple PMs were scheduled on the same machine as opposed to distributing them evenly, making upload speed of a single machine the critical bottleneck. This likely occurred because we opted to make requests per user as opposed to

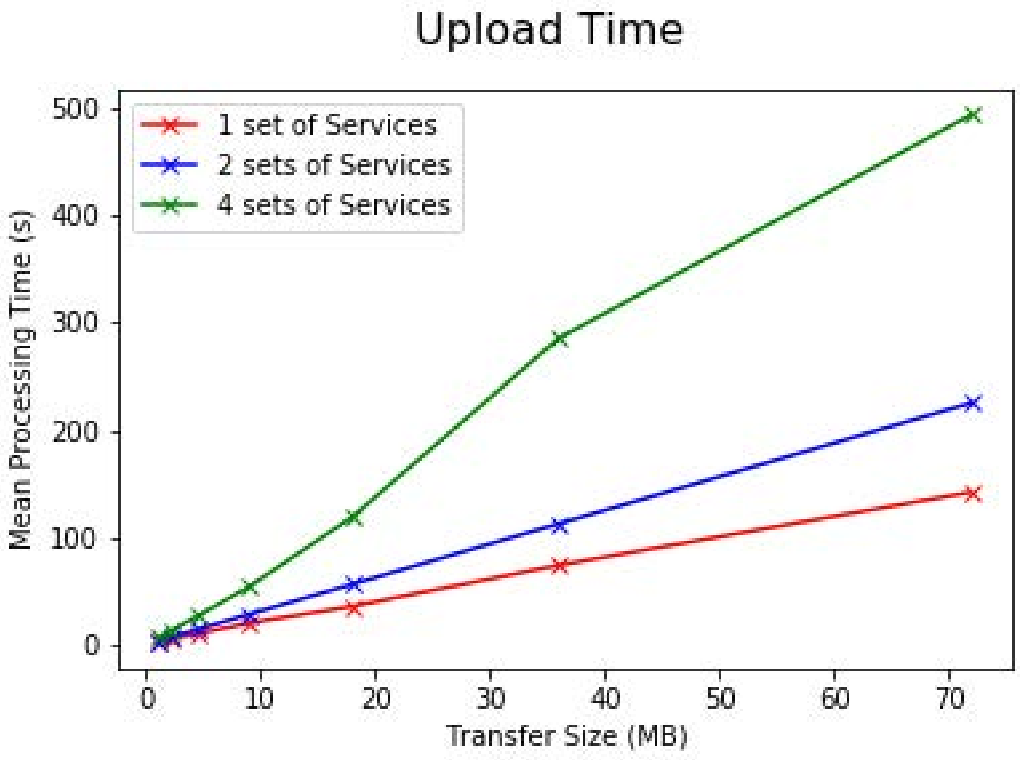

Figure 6.1: Time taken to upload files from one client machine outside the cloud to a set of PMs running in the cloud.

trying to achieve optimal scheduling.

6.3 Micro Benchmark Discussion

The most significant takeaway from the micro benchmarks used are that Kubernetes pods are not a reasonable fundamental building block for UPC. While our data transfer and service cost benchmark shows reasonable performance, the significant deletion time of each pod is unacceptable for services that need to be dynamically spawned. This problem could be potentially avoided by never deleting pods. However, pods are designed to be able to quickly scale, so as discussed in Section 6.1, relatively few distinct pods can launch at one time, to allow support for many replica of each pod.

To properly introduce UPC into Kubernetes, different building blocks and different methods of transitioning between users are necessary. One option is rather than providing each user their own pod, pods can be reused across users after some steps to detach the existing pod from an user state. This may require wiping certain sections of memory, removing mounts, or relying on stateless services when possible. Any of these steps will have addi-

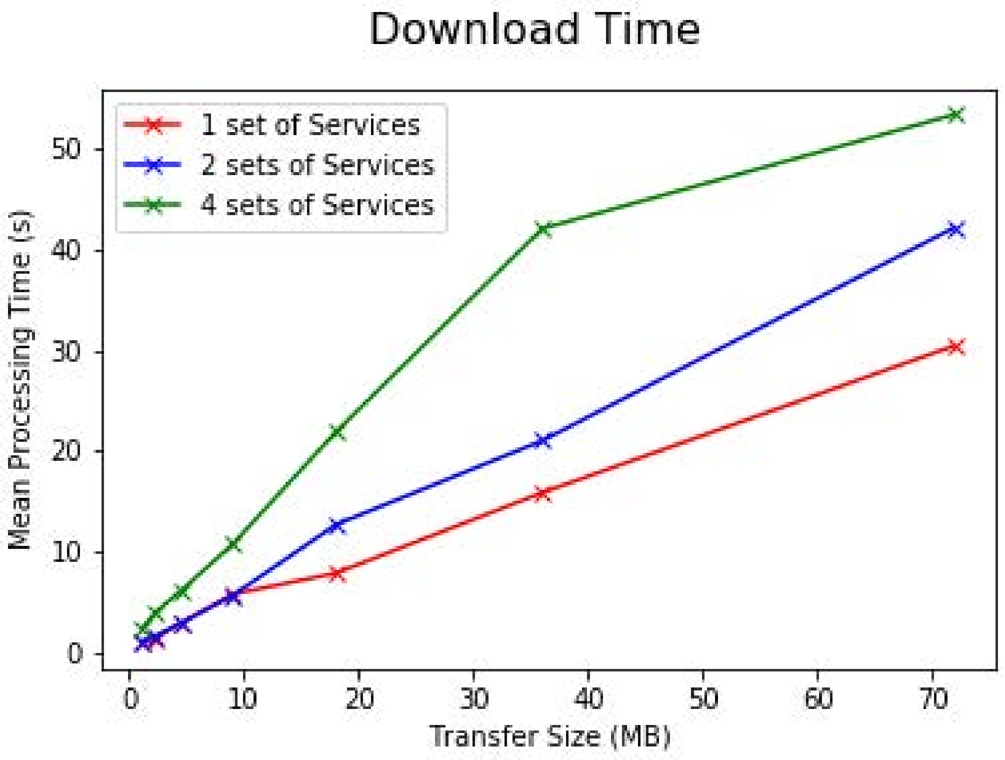

Figure 6.2: Time taken to download files from a set of PMs running in the cloud to a client machine outside the cloud.

tional overhead and add to the software complexity of a UPC architecture. An alternative option would be to modify Kubernetes and allow for containers as the fundamental building blocks of UPC. Doing so would UPC to leverage container pausing, which allows inactive containers to be transferred to non-volatile storage, allowing a greater number of concurrent containers to be spawned on one machine. Kubernetes currently does have support for allowing administrators to specify that they want an entire pod or subset of pod components to be paused, but such a change may allow support for a greater number of concurrent users or even allocating each user their entire software stack. This is likely extremely beneficial to UPC because we expect that most user services will be inactive the vast majority of the time. Finally, it would likely prove beneficial to specify a pod as fixed size so the system doesn't allocate unnecessary resources to quickly spawns additional pods that will never be requested.

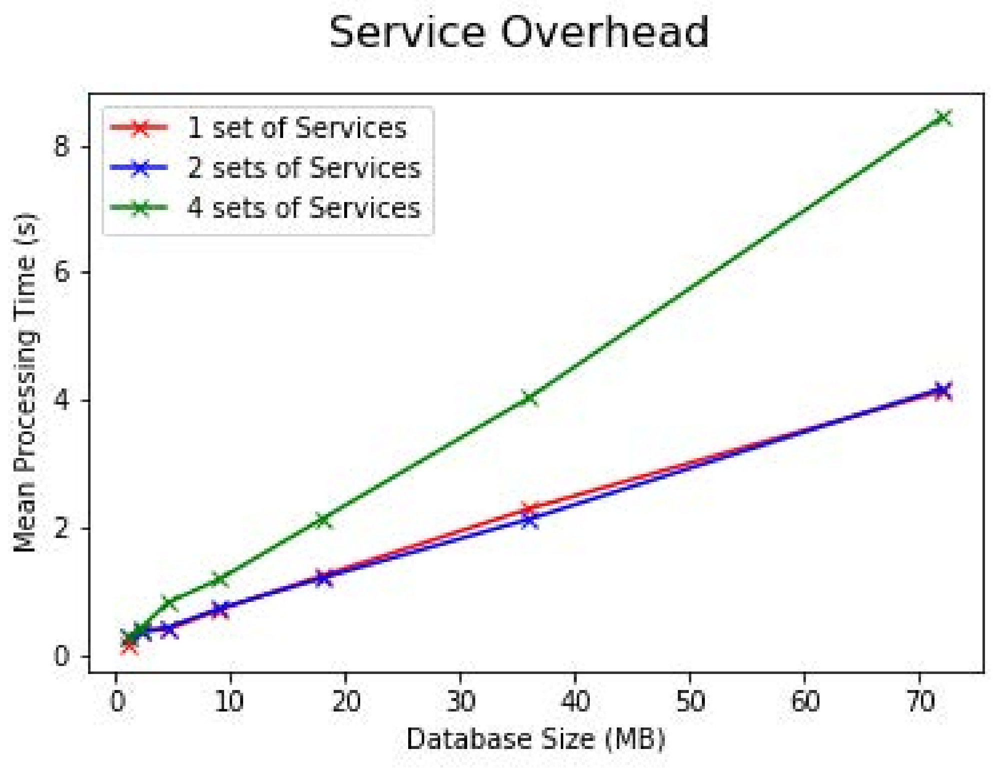

Figure 6.3: Time taken to download data from a PM to another service in the cluster. This is always a fixed 1.1 KB download, so it is intended to measure a baseline for service overhead as database size increases.

6.4 E-Mission Fake Data

In order to evaluate our E-Mission modifications, we needed to produce a representative set of user travel data. While E-Mission is currently being used in existing data collection projects, that data is private to the particular studies. As a result, we generated fake trip diaries from synthetic population data for a set of users in Alameda County, California.

The process by which we constructed our synthetic data is heavily influenced by [13]. To first construct our population, we used the open source SynthPop [14] offered by Lawrence Berkeley National Laboratory. SynthPop implements the algorithm in [91], whose primary benefit is its more accurate representation of household distributions across the synthetic population. However, since the queries in which we were interested do not require detailed tracking of household dependencies, we only referred to the individual data SynthPop produced. This gave us a population of 118,000 users from within Alameda County, California based on U.S. Census Data. For our measurements, we then sampled 5000 users from this population.

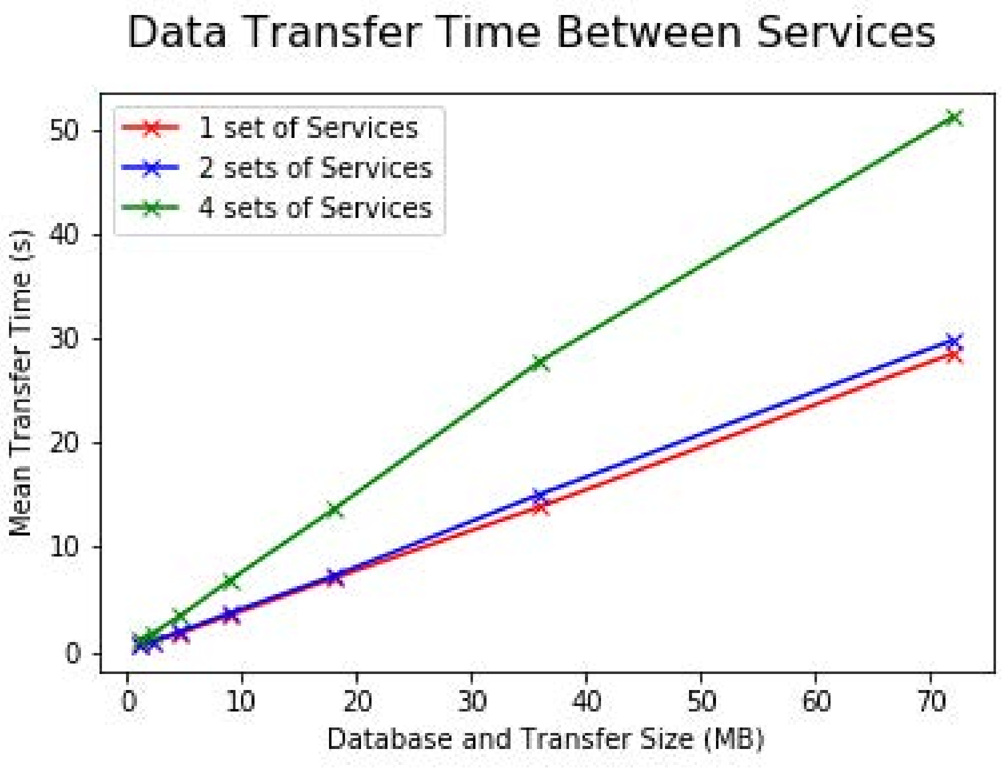

Figure 6.4: Time taken to transfer the entire database contents from a PM to another service in the cluster.

To produce a trip diary for each user, we relied on data made available by the NHTS2017. The NHTS2017 contains a set of participants with demographic information and a set of trip diaries from users around the country. These trip diaries are labeled with mode of transportation, reason for the trip, distance traveled, and time of travel, among other characteristics. To select trip diaries, we filtered the NHTS2017 data based upon the student and employment status of our synthetic user and then randomly selected a trip diary from the remaining NHTS2017 participants.

To construct usable data, we opted to adopt a tour model, in which any user's set of trips both begins and ends at their home location. As a result, we removed any NHTS2017 participants who did not follow this model. We also filtered NHTS2017 to remove participants who traveled by airplane and boat as we could not represent these modes of transportation. Finally, we also assumed that each user has one home and work location. This required us to remove any NHTS2017 participants who took a trip from home to home or work to work.

After we acquired trip diaries, we mapped the behaviors to actual locations using the Overpass API [15] to query Open Street Maps [16]. For each user, we first assigned a random home from the set of residential addresses in the City of Berkeley. Then to get the next location, we converted the reason provided for taking a trip into location types that could match that reason. For example, for any trip for religious reasons, we would search for houses of worship. We then queried Open Street Maps using the location types and selected the location which was closest to the distanced provided by the NHTS2017 trip diary. The two exceptions to this process were trips to home and work. Any trip home always returned to the same location, regardless of distance. Similarly, users also have one work location. So once the initial location was selected, we again ignored distance for subsequent trips to work. However, if there was a trip between home and work, we always used that relationship to generate the work location because the most common trip in the survey is between home and work. Our code implementing this process is available for use or review [17].

Once we had the endpoints for our trips, we constructed synthetic E-Mission data using the E-Mission fake data generator [18]. This was constructed primarily by Alvin Ghouas and then modified to support our input format by K. Shankari. The data generator queries Open Trip Planner [65] to construct a path between our endpoints which resemble data collected on smartphones by E-Mission. We verified that our modifications allowed E-Mission to operate properly on our synthetic data when running in UPC.

Chapter 7

Future Work

7.1 Authentication

One important component of any UPC implementation is developing a sufficient mechanism for regulating access to data. This means both deciding on a sufficient model for delegating data permissions as well as ultimately integrating technologies specialized in delegated access. One challenge with selecting any model is ensuring that the model is expressive enough so users can adequately indicate what data they want applications to access (*e.g.* selecting which data to access, what time interval, *etc.*), but also ensuring that these models are simple enough so users actually may adopt policies or dedicate interest to them. Much of this work could be based around the research into smartphone permissions models surrounding IoS [54] and Android [84] operating systems. Some research [74] suggests that smartphone permissions bombard users with over one hundred permission. However, other research [55] also suggests that users tended to follow 78.7% of the recommendations they received from a Personalized Privacy Assistant. Unfortunately, that same work concluded that, despite consistent nudges recommending users change their settings, users made relatively few changes to their permissions configuration. This suggests that an adequate and expressive default permission model is probably the most important future step towards building a proper permissions model. Additionally, to better regulate these models, tools designed around authentication, such as OAuth [63] or WAVE [26], could be used in conjunction with the PM. This could allow decoupling the launching of a service with its subsequent data requests, which will ultimately allow for more detailed permissions policies and hopefully reduce some of the trust required in selected services.

7.2 Service Listings

Throughout much of this work we discussed a user being able to add services, but have shied away from providing a detailed description of this process. Our current work assumes the existence of a list of commonly available services deployed as docker images. We believe docker images are an ideal choice for deployment in a cloud service environment, but our current implementation relies on either Docker Hub [37], Google Container Registry [19], or locally tagged images, none of which we consider ideal. The problem with all of these platforms are that they provide no guidance for determining the quality of a container. We believe that UPC service listing site(s) should be based on the model presented by Github Marketplace [20]. Github Marketplace is a platform provided by Github [21] which aims to provider developers with small applications for integration into their larger projects. Developers can directly publish their applications, possibly making their applications available only to other developers for a fee. Most notably, Github Marketplace applications are classified as either reviewed and approved by Github employees or unverified. This verification model could extend well to a UPC setting where the review process would analyze whether any services correctly apply UPC dependent features, such as handling differentially private queries properly or ensuring that services are not written to store user data. One appeal of designing a service listing after this model is that application developers have already been exposed to it, increasing the likelihood they will be able to effectively leverage its potential benefits for application development. It also offers a potential resolution of issues with incompletely or broken applications as expressed for IFTTT in [56]. Users and developers can have confidence that reviewed applications will work properly and are private. By registering with the service listing, users can also receive notifications when any issues are discovered.

An additional challenge that needs to be addressed in the future is illustrating the process by which users can familiarize themselves with a set of available services and ultimately opt to trust them. Users potentially have less technical expertise and, unlike application developers; may not be familiar with Github Marketplace. The search features provided for users and developers may also need to differ, as developers likely have greater insights into the components needed for applications, whereas users may only understand the broader function. This suggests that for users, a search should classify services by applications, and then users should be notified of each service's review status. Finally, service listings also produce questions about who to trust. The Github Marketplace model is promising because it places the UPC implementation architects as chiefly trusted, but users or developers may also seek verification from third-parties. The UPC open source requirement easily accommodates this, but a service listing will likely have to answer questions about how, if at all, to display third-party reviews, to ensure users have access to information but are not overwhelmed by too many reviews.

7.3 Data Management

The decision to place users in control of their processed data in addition to their raw data prompts new questions about the life cycle of this data. While a large amount of memory is cheap for large companies, giving users control of their data confronts them with questions about if they should delete their data. Similarly, data costs potentially increase now that, in addition to raw data, users must also worry about the storage burden of holding processed data as well. This may seem like a simpler problem, as users should always be able to recreate processed data from the raw data, but opting to delete processed data could induce significant delays between applications, especially for large applications. In addition, it is unclear that raw data is even likely to be preferred by some users. For example, user experiences with Google Timelines may mean they opt to store location data in an easy-to-read processed format and may never again refer to their raw data. This significantly complicates data deletion as well because one must ask, if raw data corresponding to a certain data is deleted, should the processed data be deleted as well? One viewpoint would say that it should as the user is seeking to remove that event from their history; another would say that users may be reducing storage by opting to store only the processed form. The entire interaction between raw and complex data is also further complicated by global aggregates, as differential privacy corresponding to raw data needs to propagate down to all data that is later processed.

7.4 Deployment Alternatives

We opted to implement our UPC example using only commonly available cloud resources and technologies. We made this decision because we felt commonly available cloud resources had the fewest number of user requirements, allowed for smooth transition from a traditional cloud application, were implementable on all cloud providers, and offered reasonable trust assumptions in the process. In this section, we explore the potential advantages of adopting two different approaches: Trusted Execution Environments and user-owned hardware. Additionally, we will the reasons why we did not ultimately choose either of these deployments for our sample implementation. Ultimately, we would like to emphasize that both of these deployment alternatives are reasonable options for a UPC implementation.

Trusted Execution Environments

A Trusted Execution Environment (TEE) is a type of secure hardware designed to allow computation to occur without exposing the contents of the computation. Notable examples of TEEs are Intel SGX and the iPhone Secure Enclave [75]. Typically, TEEs provide their security guarantees by executing applications inside a secure enclave with specialized hardware. This enclave offers a few advantages for keeping computation private, but for our use case the biggest advantage is that when enclave memory is swapped to non-volatile storage the data is always encrypted, including any data contents paged to memory. As a result, one major benefit of transitioning the PM to run within a secure enclave is that it eliminates any concerns of a temporary secret being moved to non-volatile storage and then being visible. Another advantage enclaves offer is that they authenticate their contents, providing an alternative for service authentication.

The biggest concern with TEEs is that constructing applications for them is significantly more difficult than creating a docker image. Since Intel SGX is the most widely available implementation of a TEE and has the most resources dedicated to it, we will primarily base our assessments on challenges associated with Intel SGX. While Intel provides an SDK [53] for users to write C applications within hardware enclaves, it is very limiting. Other projects, such as SCONE [28] and Graphene [86] offer support for more language options, including scripting languages, and have added support for deploying applications in docker containers. However, attempting to implement UPC inside of either project we encountered issues. For example, when we were developing, we found that remote attestation, the process of validating an enclave located on a remote machine, was not easily supported by either project. We found another project that was working on it [34] using Graphene, but this was restricted to a subset of Graphene that only supported C code. Since retooling our entire code base into C was not feasible, we realized that while in the coming years Intel SGX may be a reasonable target for a UPC architecture, the tooling that we seek to adopt may not be well enough developed yet. Additionally, Intel SGX compatibility puts additional constraints on provisioning servers. While the number of cloud providers that offer SGX has increased from originally just IBM Cloud [51], to include, among others, Microsoft Azure [33] and Alibaba Cloud [23], Intel SGX is still not a standard feature offered by all cloud providers [30].

Personal Servers

The biggest point of contention for most users will likely be transferring keys to the PM. If software failures cause the keys streamed to the PM to become externally visible, then all of the users' secrets and data stored in the cloud become accessible by the cloud provider or an attacker. One alternative to avoid this issue is for users to purchase personal servers or hardware. In this model, rather than treating the PM as a regular service, the PM will be a special service whose address the user can provide to the service router. Then, any computation that makes requests for storage will first route through the user's PM. The primary advantages of this approach are that a user no longer needs to stream keys from their devices and can place all key information directly in the PM. Alternatively, if a user still wants to have device specific keys, then transferring keys to the PM can be done locally. This solution will likely offer a hardware storage device very similar to the MyCloud hardware solution. The necessary provisions for such a solution could be minimal, needing only to provide external access, the ability to decrypt and encrypt data, the ability to verify any restrictions on data imposed by the user, and enough memory to store user keys. In addition, there could also be a performance advantage because having a server allows a user to have one long-term PM rather than spawning a new PM every time a service request is issued. The primary reasons we opted not to require a hardware addition were that it places an extra financial burden on the user and that users would be responsible for configuring and managing the hardware. While the requirements for device capabilities are minimal, servers are notoriously difficult to manage, and we cannot confidently conclude that a reduced set of requirements will alleviate this burden. One could also move other parts of the UPC architecture to privately own hardware, but we do not believe there would be a significant advantage provided by doing so.

7.5 User Compensation for Data

One possible motivation for having users participate in aggregation is to pay users directly for their data. We have not given this much thought, but students at TU Munich are exploring implementing a payment system using Ethereum smart contracts [89]. While the project is still largely under construction, the relevant modification as it pertains to UPC, is rather than simply authorizing services to access user data, users are offered payment proportional to the amount of privacy consumed by each transaction. Users would still need to approve each service, but approving a service would no longer allow it to use a user's data as much as desired, which is a relevant concern with fixed privacy budgets.

Chapter 8

Conclusion

In this work we introduced UPC, an architecture which prioritizes letting data owners decide exactly what operations can be performed on their data. Users can still participate in large projects because UPC has support for differential privacy. The computation layer of UPC is deployed on common cloud technologies, leveraging a micro service style architecture and Linux containers. We provided a sample implementation which operates inside of Kubernetes and converted an existing research project to operate in our new architecture. Our experiments suggest that while converting a centralized application to a distributed application has increased costs, proper cluster management should allow many popular applications to function without significant delays.

UPC offer many possible design trade-offs with respect to how applications are specified, how data is stored, and what technologies are used. We have not fully explored this design space and believe that many promising variations exist. However, regardless of the exact criteria for implementing UPC or even if future deployments match the UPC definition, it seems clear that providing users control over how their data is processed is an essential component of modern computing that needs to be adopted.

Bibliography

[1] URL: https://gdpr-info.eu/.

[2] URL: https://www.facebook.com.

[3] URL: https://www.google.com/maps/timeline.

[4] URL: https://www.gedmatch.com.

[5] URL: https://www.ancestry.com/.

[6] URL: https://www.23andme.com/.

[7] URL: https://e-mission.eecs.berkeley.edu/#/home.

[8] URL: https://aws.amazon.com/lambda/.

[9] URL: https://redis.io/.

[10] URL: https://aws.amazon.com/s3/.

[11] URL: https://pymongo.readthedocs.io/en/stable/.

[12] URL: https://cloud.google.com/kubernetes-engine.

[13] URL: https://www.simunto.com/matsim/tutorials/eifer2019/slides_day2.pdf.

[14] URL: https://github.com/LBNL-UCB-STI/synthpop.

[15] URL: https://github.com/drolbr/Overpass-API.

[16] URL: https://www.openstreetmap.org.

[17] URL: https://github.com/njriasan/e-mission-thesis-fake-data.

[18] URL: https://github.com/e-mission/em-dataload.

[19] URL: https://cloud.google.com/container-registry.

[20] URL: https://github.com/marketplace.

[21] URL: https://github.com/.

[22] *2017 National Household Travel Survey, U.S. Department of Transportation, Washington, DC.* Tech. rep. Federal Highway Administration, 2017. URL: https://nhts.ornl.gov/.

[23] *Alibaba Cloud.* URL: https://us.alibabacloud.com/.

[24] Keith Allen, Jason Hanna, and Cheri Mossburg. "Police used free genealogy database to track Golden State Killer suspect, investigator says". In: *CNN* (Apr. 2018). URL: `https://www.cnn.com/2018/04/26/us/golden-state-killer-dna-report/index.html`.

[25] Anati et al. *Innovative Technology for CPU Based Attestation and Sealing Ittai.* 2013.

[26] Michael P Andersen et al. "WAVE: A Decentralized Authorization Framework with Transitive Delegation". In: *28th USENIX Security Symposium (USENIX Security 19).* Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1375–1392. ISBN: 978-1-939133-06-9. URL: `https://www.usenix.org/conference/usenixsecurity19/presentation/andersen`.

[27] *App Engine — Google Cloud.* URL: `https://cloud.google.com/appengine`.

[28] Sergei Arnautov et al. "SCONE: Secure Linux Containers with Intel SGX". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16).* Savannah, GA: USENIX Association, Nov. 2016, pp. 689–703. ISBN: 978-1-931971-33-1. URL: `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov`.

[29] *AWS Elastic Beanstalk.* URL: `https://aws.amazon.com/elasticbeanstalk/`.

[30] Ayeks. *ayeks/SGX-hardware.* Oct. 2019. URL: `https://github.com/ayeks/SGX-hardware`.

[31] Bradford Betz. *Facebook settles in Illinois for $550M in rare privacy law.* Feb. 2020. URL: `https://www.foxnews.com/tech/facebook-illinois-privacy-law-settlement`.

[32] *California Consumer Privacy Act (CCPA).* Apr. 2020. URL: `https://oag.ca.gov/privacy/ccpa`.

[33] *Cloud Computing Services: Microsoft Azure.* URL: `https://azure.microsoft.com/en-us/`.

[34] Cloud-Security-Research. *cloud-security-research/sgx-ra-tls.* URL: `https://github.com/cloud-security-research/sgx-ra-tls`.

[35] *Cozy Cloud - A Personal Cloud to gather all your data.* URL: `https://cozy.io/en/`.

[36] Damien Desfontaines. *Local vs. global differential privacy - Ted is writing things.* URL: `https://desfontain.es/privacy/local-global-differential-privacy.html`.

[37] *Docker Hub.* URL: `https://hub.docker.com/`.

[38] Cynthia Dwork. "Differential Privacy". In: *Automata, Languages and Programming.* Ed. by Michele Bugliesi et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–12. ISBN: 978-3-540-35908-1.

[39] Cynthia Dwork. "Differential Privacy: A Survey of Results". In: *Theory and Applications of Models of Computation.* Ed. by Manindra Agrawal et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–19. ISBN: 978-3-540-79228-4.

[40] Cynthia Dwork and Aaron Roth. "The Algorithmic Foundations of Differential Privacy". In: *Found. Trends Theor. Comput. Sci.* 9.3–4 (Aug. 2014), pp. 211–407. ISSN: 1551-305X. DOI: 10.1561/0400000042. URL: https://doi.org/10.1561/0400000042.

[41] *eCryptfs*. URL: https://www.ecryptfs.org/.

[42] *Effortless security from anywhere.* URL: https://www.lastpass.com/.

[43] *Empowering App Development for Developers.* URL: https://www.docker.com/.

[44] *Enigma*. Mar. 2020. URL: https://blog.enigma.co/safetrace-privacy-preserving-contact-tracing-for-covid-19-c5ae8e1afa93.

[45] Jonas Fritzsch et al. *From Monolith to Microservices: A Classification of Refactoring Approaches*. July 2018.

[46] Carmit Hazay and Yehuda Lindell. "A Note on the Relation between the Definitions of Security for Semi-Honest and Malicious Adversaries." In: *IACR Cryptology ePrint Archive* 2010 (Oct. 2010), p. 551. DOI: 10.1007/978-3-642-14303-8_3.

[47] Scott Hendrickson et al. "Serverless Computation with OpenLambda". In: *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'16. Denver, CO: USENIX Association, 2016, pp. 33–39.

[48] *Heroku*. URL: https://www.heroku.com/.

[49] Kashmir Hill and Heather Murphy. "Your DNA Profile is Private? A Florida Judge Just Said Otherwise". In: *New York Times* (Nov. 2019). URL: https://www.nytimes.com/2019/11/05/business/dna-database-search-warrant.html.

[50] *IaaS vs PaaS vs SaaS: What You Need to Know Examples (2018)*. Jan. 2020. URL: https://www.bigcommerce.com/blog/saas-vs-paas-vs-iaas/.

[51] *IBM Cloud*. URL: https://www.ibm.com/cloud.

[52] Ifttt. *IFTTT*. URL: https://ifttt.com/.

[53] Intel. *intel/linux-sgx*. URL: https://github.com/intel/linux-sgx.

[54] *iOS 13*. URL: https://www.apple.com/ios/ios-13/.

[55] Bin Liu et al. "Follow My Recommendations: A Personalized Privacy Assistant for Mobile App Permissions". In: *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*. Denver, CO: USENIX Association, June 2016, pp. 27–41. ISBN: 978-1-931971-31-7. URL: https://www.usenix.org/conference/soups2016/technical-sessions/presentation/liu.

[56] James A. Martin and Matthew Finnegan. *What is IFTTT? How to use If This, Then That services*. Jan. 2019. URL: https://www.computerworld.com/article/3239304/what-is-ifttt-how-to-use-if-this-then-that-services.html.

[57] Yves-Alexandre De Montjoye et al. "openPDS: Protecting the Privacy of Metadata through SafeAnswers". In: *PLoS ONE* 9.7 (Sept. 2014). DOI: 10.1371/journal.pone.0098790.

[58] *My Cloud*. URL: `https://www.mycloud.com/#/`.

[59] *Mydex CIC*. URL: `https://mydex.org/`.

[60] Nextcloud. *Nextcloud*. URL: `https://nextcloud.com/`.

[61] Nguyen. *Understanding Differential Privacy*. July 2019. URL: `https://towardsdatascience.com/understanding-differential-privacy-85ce191e198a`.

[62] S. O'Dea. *U.S. smartphone sales by year 2019*. Feb. 2020. URL: `https://www.statista.com/statistics/191985/sales-of-smartphones-in-the-us-since-2005/`.

[63] *OAuth 2.0*. URL: `https://oauth.net/2/`.

[64] *of-All-Things*. URL: `https://www.hubofallthings.com/`.

[65] *OpenTripPlanner*. URL: `https://www.opentripplanner.org/`.

[66] *Password Manager App for Home, Mobile, Business*. URL: `https://www.dashlane.com/`.

[67] *Password Manager for Teams, Businesses, Enterprises: Zoho Vault*. URL: `https://www.zoho.com/vault/`.

[68] *Password Managers: Under the Hood of Secrets Management*. Feb. 2019. URL: `https://www.ise.io/casestudies/password-manager-hacking/`.

[69] *Personal Server at Home*. URL: `https://freedombox.org/`.

[70] *Production-Grade Container Orchestration*. URL: `https://kubernetes.io/`.

[71] *Protect Your Passwords*. URL: `https://www.roboform.com/`.

[72] *Python Web Framework*. URL: `https://bottlepy.org/docs/dev/`.

[73] Christopher Riederer et al. "For Sale: Your Data: By: You". In: *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*. HotNets-X. Cambridge, Massachusetts: Association for Computing Machinery, 2011. ISBN: 9781450310598. DOI: `10.1145/2070562.2070575`. URL: `https://doi.org/10.1145/2070562.2070575`.

[74] Paul Sawers. *Android Users Have an Average of 95 Apps Installed on Their Phones*. Aug. 2014. URL: `https://thenextweb.com/apps/2014/08/26/android-users-average-95-apps-installed-phones-according-yahoo-aviate-data/#gref,%202014.Accessed:2016-02-01`.

[75] *Secure Enclave overview*. URL: `https://support.apple.com/guide/security/secure-enclave-overview-sec59b0b31ff/web`.

[76] Adi Shamir. "How to Share a Secret". In: *Commun. ACM* 22.11 (Nov. 1979), pp. 612–613. ISSN: 0001-0782. DOI: `10.1145/359168.359176`. URL: `https://doi.org/10.1145/359168.359176`.

[77] KALYANARAMAN SHANKARI. "e-mission: an open source, extensible platform for human mobility systems". PhD thesis. EECS Department, University of California, Berkeley, Dec. 2019. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-180.html`.

[78] David Siegel. *Understanding The DAO Attack*. June 2016. URL: `https://www.coindesk.com/understanding-dao-hack-journalists`.

[79] Vikram Sreekanti et al. *Cloudburst: Stateful Functions-as-a-Service*. 2020. arXiv: `2001.04592 [cs.DC]`.

[80] John Sullivan. "An Approach to Privacy Preserving Queries for Spatio-Temporal Data". MA thesis. EECS Department, University of California, Berkeley, May 2019. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-69.html`.

[81] Dan Swinhoe. *The 15 biggest data breaches of the 21st century*. Apr. 2020. URL: `https://www.csoonline.com/article/2130877/the-biggest-data-breaches-of-the-21st-century.html`.

[82] *Telecommunications*. URL: `https://ihsmarkit.com/industry/telecommunications.html`.

[83] *The most popular database for modern apps*. URL: `https://www.mongodb.com/`.

[84] *The platform pushing what's possible*. URL: `https://www.android.com/`.

[85] Hien To, Gabriel Ghinita, and Cyrus Shahabi. "A Framework for Protecting Worker Location Privacy in Spatial Crowdsourcing". In: *Proc. VLDB Endow.* 7.10 (June 2014), pp. 919–930. ISSN: 2150-8097. DOI: `10.14778/2732951.2732966`. URL: `https://doi.org/10.14778/2732951.2732966`.

[86] Chia-che Tsai, Donald E. Porter, and Mona Vij. "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX". In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, July 2017, pp. 645–658. ISBN: 978-1-931971-38-6. URL: `https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai`.

[87] Kurt Wagner. "This is how Facebook uses your data for ad targeting". In: *Recode* (Apr. 2018). URL: `https://www.vox.com/2018/4/11/17177842/facebook-advertising-ads-explained-mark-zuckerberg`.

[88] *What is Solid?* URL: `https//solid.mit.edu/`.

[89] Gavin Wood. "Ethereum: A secure decentralised generalised transaction ledger". In: ().

[90] Meng Xiaofeng and Ci Xiang. "Big Data Management: ConceptsTechniques and Challenges". In: *Journal of Computer Research and Development* 50.1, 146 (2013), p. 146. URL: `http://crad.ict.ac.cn/EN/abstract/article_715.shtml`.

[91] Xin Ye et al. *Methodology to Match Distributions of Both Household and Person Attributes in Generation of Synthetic Populations*. 2009.

[92] Chuan Zhao et al. “Secure Multi-Party Computation: Theory, Practice and Applications”. In: *Information Sciences* 476 (Oct. 2018). DOI: `10.1016/j.ins.2018.10.024`.