RideGuard: Towards Privacy-Preserving Analytics for Mobility Applications

Arun Sundaresan

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-44
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-44.html

May 1, 2024

RideGuard: Towards Privacy-Preserving Analytics for Mobility Applications

by Arun Sundaresan

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:

Professor Sylvia Ratnasamy
Research Advisor

April 24, 2024

(Date)

****

Associate Professor Scott Moura
Second Reader

April 26, 2024

(Date)

Abstract

RideGuard: Towards Privacy-Preserving Analytics for Mobility Applications

by

Arun Sundaresan

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Sylvia Ratnasamy, Chair

In this paper, we present RideGuard, a tool to enable the calculation of safety indices for micromobility while preserving rider privacy. We first present a method to preserve k-anonymity over user queries to third-party location-based services (LBSs), followed by an examination of the Waffle protocol for privacy-preserving operations on key-value stores and its applications to RideGuard. Finally, we detail how to combine both of these mechanisms into a unified proxy that can calculate safety indices. We also examine the privacy and performance guarantees offered by our unified proxy. We highlight how RideGuard is built with flexibility in mind, aiming to support as many safety indices as possible, given that geographic areas can be affected by drastically different safety-related factors.

Contents

Acknowledgments

First, I want to thank Sylvia Ratnasamy and Silvery Fu for their support throughout my time in the NetSys Lab. I would also like to thank Scott Moura for broadening my horizons and inspiring me to pursue research areas I would never have considered before. Finally, and most importantly, I would like to express my utmost gratitude to my family, whose unwavering belief fueled me throughout this journey.

Chapter 1

Introduction

1.1 Status Quo

Micromobility, the use of lightweight vehicles like e-bikes and scooters, has gained significant popularity in recent years. While micromobility offers important benefits, including increased convenience and reduced environmental footprint, it also brings certain safety risks that need to be addressed. In 2021 alone, there were 80,000 emergency medical visits attributed to micromobility accidents[14]. The US electric scooter market is expected to grow 16.7% year-over-year from 2023 to 2030[12], and with this growth comes an influx of new riders, meaning the risk of using a micromobility device is amplified. Many of these accidents are caused because micromobility devices leverage existing infrastructure (i.e. bike lanes, off-street paths, sidewalks) intended for bicycles and pedestrians, leading to overcrowded environments unsuitable for micromobility usage. Recent studies have shown that collisions with other vehicles and terrain quality rank as key factors in causing micromobility accidents. Collisions with other vehicles or pedestrians (caused by, e.g. overcrowding) caused up to 43% of accidents, while factors like hilly terrain and pavement quality caused up to 6%[7][5]. Often, these accidents are indirectly affected by routing apps recommending that riders go near areas with high traffic or poor terrain quality. As a result, many potential riders are deterred[3] from using micromobility devices due to safety concerns.

System operators and governments have attempted to implement programs involving safety training, helmet sharing, and modifications to the micromobility devices themselves (such as self-balancing). In spite of these efforts, micromobility-related accidents continue to increase in frequency. Part of the continued increase may be explained by the micromobility data ecosystem's lack of information regarding safety. Currently, companies that own and operate fleets of micromobility devices such as Lime, Bird, and Bolt, use standards such as MDS and GBFS to generate data records. This information is generated by hardware built into the micromobility devices and focuses primarily on basic information about vehicles and their locations. For example, MDS includes records for trips, stops, and vehicle status (including factors like battery level). Hence, to improve rider safety, better routing is needed.

Improved approaches to routing for micromobility devices can help to reduce the number of accidents by having riders avoid areas likely to cause an accident. However, generalized routing apps such as Google Maps and specialized ones like ScootRoute compute routes using data from a combination of mapping services. These apps do not consider factors related to safety while computing routes and do not offer a way for riders to compare different routes based on safety (instead, optimizing purely for time). For example, Google Maps and ScootRoute do not take rider experience into account, but a study in Austin found that 63% of injured riders on e-scooters had previously used them 9 or fewer times[1]. Additionally, the existing approaches do not consider factors such as crowding, terrain quality, and accident history along the route.

At the same time, concerns have arisen regarding privacy preservation when users engage with location-based services, principally surrounding mapping services. Currently, the main approaches used for micromobility routing are generalized apps like Google Maps and specialized ones like ScootRoute. These apps consume data from external mapping services and crowdsourcing from other users. ScootRoute in particular takes into account user preferences about proximity to busy roads and tolerance for hills, among other factors. Some system operators have integrated a routing service that uses external mapping services into their user experience. For example, Lime partnered with CityMapper[2]. These mapping services can come in the form of GTFS when dealing with public transit, or include real-time information about crowding or closures. Of these features, crowding and user preferences are the only ones that have a close relationship with safety. Additionally, both Google Maps and ScootRoute present an incomplete picture to the rider, with Google Maps neglecting safety information altogether, and ScootRoute only returning a single route that conforms to user preferences.

1.2 RideGuard's Contributions

As mentioned before, safety-aware routing is vital to improve safety. In this paper, we present RideGuard, a tool to enable safety-aware routing for micromobility while preserving privacy over rider data from untrusted mapping services. RideGuard is implemented as a proxy between riders and untrusted services, and leverages a variant on the Sybil[13] protocol to achieve k-anonymity for location queries. While prior literature has focused on designing location-based services (LBSs) that process user information in a way that preserves privacy, we instead focus on the case where we cannot change the workings of the LBSs that supply information vital for safety. K-anonymity ensures that we can send data relevant to the rider's real route while preserving privacy by sending information about similar fake routes. It also allows rider privacy to be preserved in the face of both untrusted route generators and untrusted third-party APIs that supply safety information since we need only send all points from both the real and fake routes.

For storing rider records containing safety-related information (in a potentially untrusted datastore), we employ the Waffle protocol[8]. While encryption under an IND-CPA-secure

scheme ensures data cannot be read directly by an adversary monitoring the datastore, access patterns and query patterns can still reveal unwanted information about a rider[9]. Waffle uses fake queries on real objects and fake queries on dummy objects resembling real ones to obscure a rider's true access patterns. Waffle assumes a key-value store, so we detail a method to adapt RideGuard to Waffle. Since it is impractical in Waffle to ensure that all access patterns over all keys are perfectly uniform, we use Waffle's alpha-beta uniformity privacy model to ensure that the access patterns are sufficiently similar and all have the same sequence of actions taken on them. We leverage data collected from riders to rate different routes based on various safety-related factors, letting the user decide which one to use. This approach combines the best of Google Maps and ScootRoute, allowing the user to take their own preferences when selecting a route, while providing them with information about the risks of each route. In contrast with existing MDS and GBFS-based approaches, we leverage phones mounted to micromobility devices to collect data, providing greater interoperability than using vehicle hardware. As a result, RideGuard can work with any micromobility vehicle regardless of the owner (including privately-owned micromobility vehicles). We aim to make RideGuard as flexible as possible, in order to be usable in areas with different sets of factors affecting micromobility safety.

In the remainder of this thesis, we first devise a variant on the Sybil protocol to preserve k-anonymity in location queries to third-party services. Then, we discuss how to adapt RideGuard to the Waffle protocol. Finally, we examine the performance of RideGuard on a real-life micromobility workload.

Chapter 2

Query Privacy

2.1 System Model

To improve rider safety, RideGuard needs to support riders querying various location-based services (LBSs) for information about the conditions of different routes. This includes both route generators (such as Google Maps) and APIs that provide safety information (such as TomTom or Placer). Additionally, RideGuard compensates for the shortcomings of existing mobility data standards by collecting information shown to affect rider safety, for example, rider experience and events such as swerves or sudden stops. To this end, RideGuard aims to protect rider privacy from two main adversaries: LBSs and untrusted datastores.

In keeping with prior literature such as Arx[10], RideGuard assumes riders are trusted entities who normally send their real origin and destination (OD pair) to honest-but-curious LBSs. RideGuard offers a trusted proxy sitting between users and LBSs that leverages prior observed traffic patterns to preserve user privacy when location data are sent to the LBS. While multiple external APIs are necessary to provide a comprehensive view of rider safety in practice, we assume that all LBSs collude and can be modeled as a single adversary that receives all location data submitted by RideGuard (and indirectly, the rider).

Since RideGuard requires additional data collection for fields not included in mainstream mobility data standards, it also introduces an adversary at the datastore where user-submitted records are kept. We assume this adversary is honest-but-curious and can observe and remember all access patterns and query patterns on the datastore. Hence, the RideGuard proxy leverages fake queries to prevent the adversary from inferring a rider's true route based on access patterns. Further, we assume that LBSs and the datastore adversary have knowledge of traffic patterns and can access any publicly available information regarding micromobility. Notably, we only consider passive adversaries, since actively malicious adversaries could modify or delete information in the database or provide deliberately incorrect routes.

2.2 Query Privacy: Related Work

While prior literature has devoted a great deal of attention towards designing services that preserve user privacy, relatively little work has focused on preserving user privacy without changing the design of the LBS. Hence, there are two main projects related to query privacy that RideGuard draws upon, where each one uses a different privacy model.

The first is SybilQuery[13], which uses a "query obfuscation" approach to achieve k-anonymity for the rider's OD pair and route. By sending k-1 fake routes along with the real one, SybilQuery aims to limit the adversary's ability to guess the real route to be no better than guessing randomly. SybilQuery is designed to operate on a specific geographic region (the San Francisco Bay Area was used in the paper) and uses a database of historical traffic patterns in the area to divide it into separate blocks using a QuadTree[6] (more heavily used areas have smaller blocks, the authors were able to represent San Francisco with roughly 16,000 blocks based on a proprietary dataset of taxi rides).

A trusted proxy stores the QuadTree and uses it to generate fake OD pairs. This is done by calculating a "traffic density" metric for each block in the QuadTree, namely the number of trips that have an endpoint or go through the block in a given timeframe. The proxy finds blocks with similar traffic densities to the real start and end blocks and randomly picks points with a similar euclidean distance apart relative to the real OD pair. Once k-1 fake OD pairs are selected, the proxy sends the real OD pair and the k-1 fakes for queries to honest-but-curious LBSs. The authors commented that SybilQuery also uses reverse geocoding to improve the quality of generated endpoints (namely, protecting against an attack where the LBS adversary sees coordinates that clearly cannot correspond to real rides). Unfortunately, the authors implemented this using a third-party mapping service (Microsoft MultiMap), meaning the SybilQuery workflow needs to be modified for use in RideGuard.

Another paper dealing with privacy-preserving queries to routing services is ShiftRoute[15]. The ShiftRoute threat model is stricter than RideGuard's, assuming an untrusted anonymization server. ShiftRoute uses a set of points of interest (POIs) and searches for a number of POIs near the real OD points to construct a set of fake endpoints. ShiftRoute uses a linear program to calculate probabilities of using each fake OD pair to generate routes (attempting to maximize utility while preserving privacy), and aims to satisfy geo-indistinguishability, a privacy model that aims to preserve user privacy within some radius of real user locations. Specifically, given a real point, shifted points within a certain radius of the real one should have similar enough probabilities of being chosen. Crucially, perturbations can have a material impact on the route distance and usability of routing apps. For walking routes, ShiftRoute showed that perturbing the origin and destination could increase route distance by over 200 meters on average, and move the presented origin and destination by about 100 meters each from the real values, on average.

2.3 Choosing a Privacy Model

Many micromobility trips are short, with the majority in the BayWheels (a docked bikeshare service in the San Francisco Bay Area) dataset for 2023 possessing a length of under 2 kilometers. Hence, the perturbations presented by ShiftRoute impose a non-trivial increase in route distance for a rider. Perturbation poses additional problems for docked micromobility systems (such as BayWheels), where users are required to leave their vehicles at designated parking stations. ShiftRoute[15] offers no guarantee that a parking station will be located near a perturbed location. Perturbation can also adversely affect users of privately-owned micromobility devices, since their submitted origins and destinations may not correspond to locations with many PoIs, leaving the anonymizer to either use a larger radius (increasing the burden on the rider) or pick from relatively few PoIs. Additionally, perturbation risks placing users in unfamiliar locations, impacting the usability of applications built on top of it. In urban environments, for example, a perturbation of 100 meters can place the rider on a completely different street. Compensating for non-trivial perturbations in the route may require using the LBS, giving away the user's true OD pair. The main upside of ShiftRoute relative to k-anonymity is that it avoids the overhead of generating and sending k-1 extra routes to calculate a safety index. Since a key goal of RideGuard is interoperability with any micromobility device (privately-owned or shared, docked or dockless), we use k-anonymity, prioritizing rider utility above all else.

2.4 RideGuard Route Generation

Approaches such as Sybil use OD pairs that mimic the real-world distribution, but not necessarily trips that are likely to have occurred at the same time as the submitted one. Hence, a key principle of RideGuard's design is to generate routes that were likely to occur when the real route was submitted. For example, a route submitted at night would be accompanied by fake routes with a similar chance of occurring at night. The alternative is selecting K-1 fake routes with an OD pair not far removed from the real one, but this approach can leak the route the user takes if the routes between fake OD pairs are similar to the real route.

Additionally, both Sybil and ShiftRoute were conducted on car traffic data. Privacy for micromobility can be harder to accomplish because trips on shared mobility systems can start and end at designated parking areas, whose locations we need to assume are known to adversaries. Additionally, privately-owned micromobility devices do not need to follow these requirements since they do not need to be left in shared parking areas. Lime scooters, for example, may be left nearly anywhere. RideGuard's method of query obfuscation needs to cover both of these use cases (i.e., designated ODs and freeform ODs). Unlike Sybil and ShiftRoute, RideGuard needs to use information about the micromobility systems in the area to generate fake routes. To gather real traffic patterns, we use the publicly available BayWheels GBFS feed, a dataset of bikeshare trips in the San Francisco Bay Area updated

monthly (and we confine our analysis to San Francisco proper). In the rest of this paper, we assume that BayWheels GBFS data is representative of the overall micromobility patterns in San Francisco. To accommodate both dockless and docked micromobility systems, along with privately-owned micromobility vehicles, we consider two main cases: station-to-station (SS) and random-to-random (RR) trips. The SS case covers docked systems such as BayWheels, while the RR case covers privately-owned micromobility vehicles and dockless systems such as Lime, where vehicles may start trips from arbitrary locations.

To generate fake routes for the SS case, we first ask the user which docked micromobility system they are using and leverage traffic patterns between stations in that micromobility system. We first determine the closest station to the user's submitted origin and find other stations with a similar probability of being the origin at the hour the user submits their query. We then use the probabilities of a trip originating at a station to make a weighted random choice to select K-1 fake origins. Based on the traffic patterns from the BayWheels GBFS dataset, we calculate a transition matrix between stations, meaning that each station has an out-vector specifying the probabilities of different destination stations. We use these out-vectors to make a weighted random choice to select a destination for each fake origin station. The K-1 fake OD pairs are used alongside the real one to generate routes and obtain safety information. While this setup provides k-anonymity, a caveat is that real routes (along with timestamps) will eventually be released publicly, so the LBS adversary can find out which submitted routes were real at a later time.

For the RR case, we use a method much more similar to Sybil, leveraging a QuadTree to select origins. Since QuadTree nodes split into four when too many elements are inserted into them, they can capture differences in activity between areas. While Sybil uses QuadTree blocks to capture differences in traffic density, we insert BayWheels station locations into the QuadTree, since a higher density of stations reflects higher micromobility usage. By having nodes split when they contain 10 or more stations, we arrive at the following QuadTree blocks for San Francisco.

Notably, our QuadTree construction captures meaningful differences in the micromobility activity at different areas, with downtown San Francisco containing many small blocks. The blocks increase in size as we move further away, similar to Sybil. Additionally, we use far fewer blocks than Sybil (which uses 16000 such blocks) to capture differences in traffic. Sybil's blocks can become as small as 25 meters by 25 meters in the busiest areas, meaning that trips originating in a particular block can frequently resolve to the same address (or a small set of addresses) while using reverse geocoding. By Shannon's Maxim, we need to assume the LBS adversary knows about the QuadTree construction. With small block sizes like in Sybil, the adversary can narrow down an address to a very small block, and potentially use other information about the area to tell the real route from the fake ones. Using larger blocks introduces more ambiguity for the LBS adversary to handle.

To generate routes in the RR case, we use a similar method to the SS case. Given a real origin location, we find the QuadTree block containing it and pick QuadTree blocks with a similar probability of having a trip originate there in the hour the trip is submitted (using the origin probabilities to make a weighted random choice to select K-1 fake origins). To

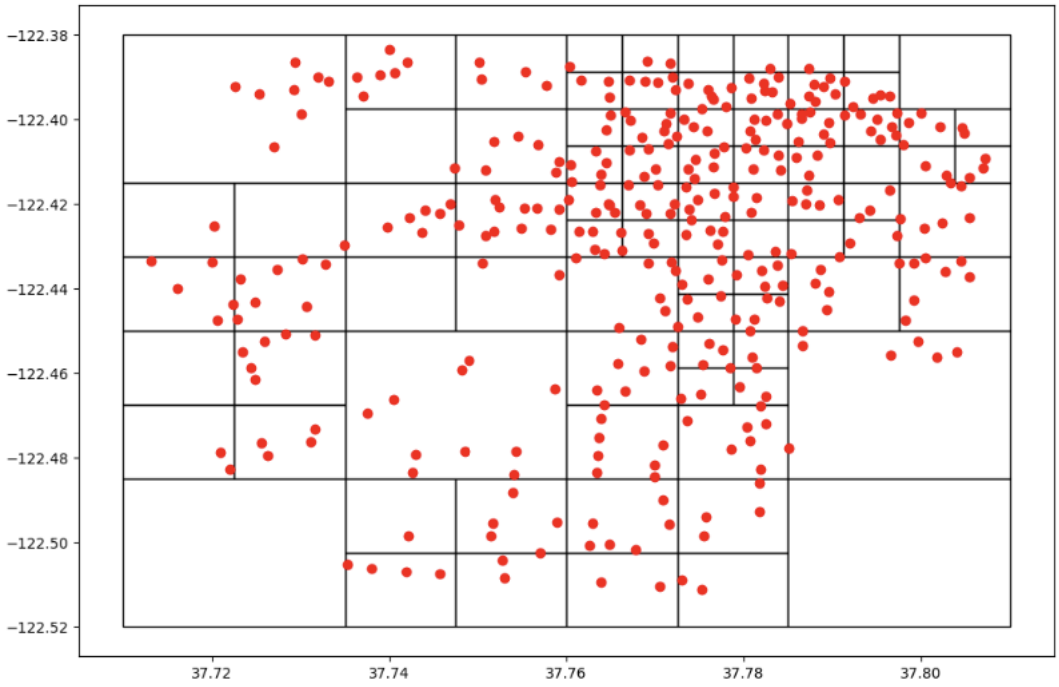

Figure 2.1: RideGuard's QuadTree Blocks in San Francisco

select destinations, we calculate a transition matrix between QuadTree blocks and use the transition probabilities to select a destination block for each origin. Within each chosen block, we randomly select a coordinate to get an OD pair for each trip.

Sybil uses reverse geocoding to improve the quality of points sent to the LBS. This defends against an attack where the LBS can eliminate possibilities in the routes it receives by ignoring the routes with OD pairs that cannot be an origin or destination. For the SS case, we accomplish reverse geocoding by simply using the coordinates of the nearest station. Sybil, on the other hand, uses Microsoft MultiMap for reverse geocoding. Since our threat model specifies that LBSs are untrusted, we instead use the OpenAddress dataset, which stores a comprehensive list of addresses in San Francisco in a 99 MB file. This approach covers both dockless micromobility systems and privately-owned micromobility vehicles while preserving rider usability since it uses addresses, as opposed to PoIs in ShiftRoute.

2.5 Implementation and Evaluation

We implemented our route generation approach using BayWheels GBFS data from the entirety of 2023 to calculate transition matrices. For a test set, we used the same dataset from January 2024. To test the indistinguishability of RideGuard's route selection from new real-world data, we devise a test where we generate fake routes for all trips from January 2024.

From the fake and real routes, we can make transition matrices (one for the fake routes, one for the real routes) and calculate cosine similarities between corresponding out-vectors (vectors of destination probabilities for a given starting point) in the real and fake matrices. Instead of using QuadTree blocks to make the transition matrices, we use geohashes of precision 6 (1.2 kilometers by 600 meters). In the event a geohash is in one transition matrix, but not in the other, we add a row of 0s. By this procedure, we obtain the following graphs for the SS and RR cases.

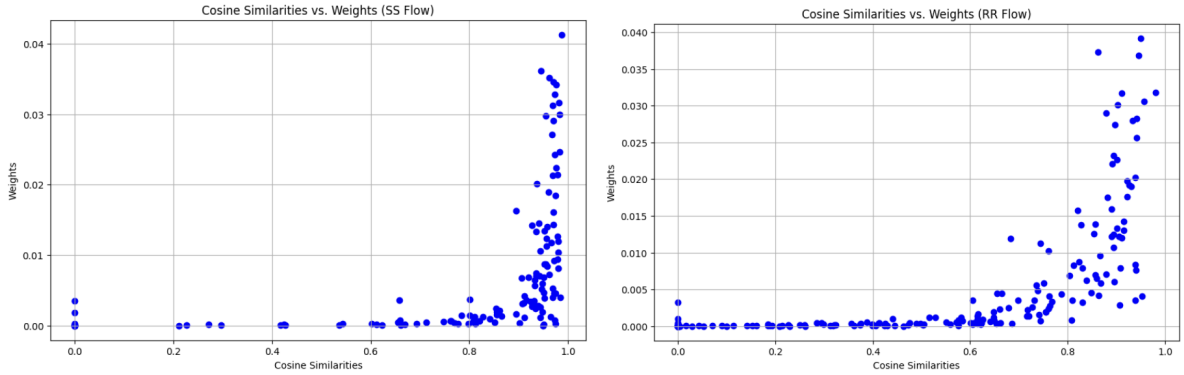

Figure 2.2: Cosine Similarities between real and fake out-vectors. X-Axis: Cosine Similarities, Y-Axis: Weight i.e. Probability of a Trip Originating From a Geohash, according to the January 2024 Dataset

We used 5 similar blocks and K=5 for both graphs (in keeping with the parameters introduced in Sybil). Both graphs exhibit dramatically higher similarities at higher weights. This is unsurprising due to the fact that geohashes with higher weights are more likely to appear in both transition matrices, thus eliminating the possibility of a zero out-vector. Additionally, having more samples increases the extent to which the data for the month are reflected in the 2023 data. Nonetheless, the fake trips in both the RR and SS flows exhibit a high level of similarity with the real ones, with 0.66% of trips in the SS flow corresponding to a cosine similarity of 0.6 or less. In the RR flows, this increases to 2.2%. A potential modification to increase cosine similarities is to be more granular in selecting which transition matrix to use to pick fake routes.

We are also concerned about the relative likelihoods of fake trips in relation to the real route in a batch. Since we assume the LBS adversary has background knowledge of traffic patterns, we need to defend against an attack where they find the real route by determining how likely all routes are (using probabilities of starting and ending in geohashes of precision 6). If the real route is reliably significantly more likely to have occurred than the fake ones, the LBS adversary can break k-anonymity. Hence, in this test, we send the fake route along with the K-1 fakes, and calculate the probabilities of each trip in the batch according to the real start and end geohashes. We then sort all routes by their probabilities of occurring and

determine the probability of a real route occurring at a particular index in the sorted list of K routes. Running this test on the RR flow generates the results of Tables 2.1 and 2.2.

Index	Probability of Real Route at Index
4 (most probable)	0.308
3	0.248
2	0.202
1	0.154
0	0.088

Table 2.1: Results for RR Flow Probabilities. 5 similar blocks, K=5.

Index	Probability of Real Route at Index
4 (most probable)	0.201
3	0.205
2	0.202
1	0.203
0	0.189

Table 2.2: Results for SS Flow Probabilities. 5 similar blocks, K=5.

While real routes are more likely to be the most probable in their batch, they can also be distributed among the less probable routes (with a probability of 0.692). Hence, RideGuard's fake routes provide a high level of indistinguishability relative to real routes. Further, when the real route was the most probable, it was separated from the next most probable route by a margin of 0.000727 on average. For the SS case, the probabilities were even more uniform, as shown in the following table.

Chapter 3

Stored Record Privacy

3.1 Stored Record Privacy: Related Work

We assume a persistent passive adversary that can see all queries to the datastore, including their access patterns. The datastore adversary receives records submitted by users, as well as requests to access safety information for different locations. If the adversary can learn the locations from the access patterns, they can possibly reidentify riders by leveraging background knowledge of traffic patterns. In RideGuard, we aim to protect against data leakage and access pattern attacks. Much prior literature about encrypted databases has focused on evaluating queries over encrypted data, relying on bespoke encryption schemes or techniques such as ORAM. However, many of these approaches either do not provide protection over access patterns (e.g., CryptDB[11], Arx[10]), or rely on different trust domains to ensure resistance to access pattern attacks (e.g., Dory[4]). Additionally, RideGuard's datastore needs to obscure a changing access pattern. For example, queries at different times of the day can access different distributions of locations.

3.2 Waffle's Privacy Model

To this end, we employ the Waffle protocol to hide access patterns in RideGuard. Waffle operates on a key-value store and adopts a similar system model to RideGuard, with a trusted proxy sitting between trusted users and a datastore monitored by an honest-but-curious persistent passive adversary. Waffle requires that all keys be known in advance.

First, to protect against access pattern attacks, we need to make sure each key at the datastore goes through the same sequence of actions, namely, being written followed by being read at most once and subsequently deleted. Second, if we insert the same key into the datastore multiple times, there is potential to leak information. Hence, we need to insert a different key into the datastore even if the key has the same meaning to the rider/proxy (specifically, we need a non-static assignment of keys used by the riders/proxy to keys inserted in the datastore). If we can achieve these two properties, in the view of the datastore

adversary, the only difference between keys is the length of time they are held in the datastore. 3.1 and 3.2 reveal the operations taken on a key throughout its lifecycle in Waffle.

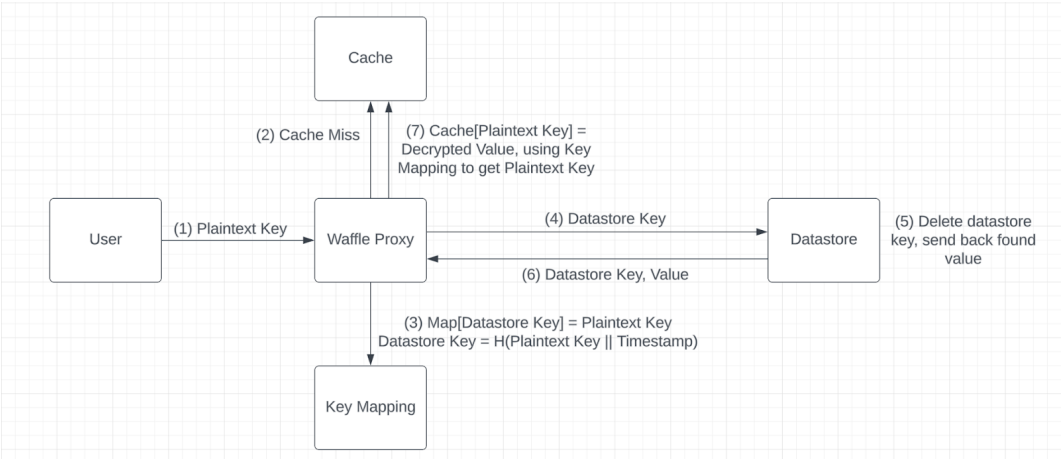

Figure 3.1: Lifecycle of a Key in Waffle when it is Initially Received

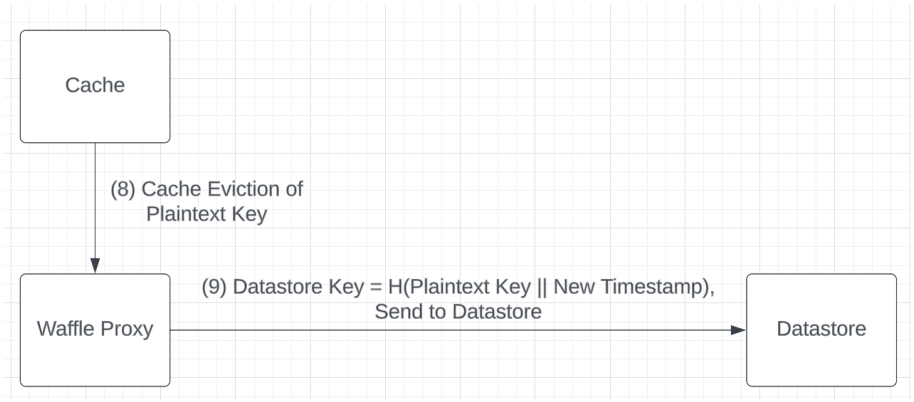

Figure 3.2: Lifecycle of a Key in Waffle when it is Evicted from the Cache

Waffle operates by batching keys users request, generating fake keys, and sending a read batch and write batch to the datastore. We refer to this procedure as a "run" of the Waffle protocol. To achieve the first property, waffle implements the notion of α-β uniformity. We are interested in two numbers: α, the maximum number of runs of the Waffle protocol between (excluding bounds) when a key is written to the datastore and read (and deleted), and β, the minimum number of runs of the Waffle protocol between a key being read from the datastore and a corresponding datastore key with a new timestamp being written back. We say α-β uniformity holds if this principle applies to all keys.

We want to minimize α since less frequently requested objects will be read sooner after they are written. Conversely, we want to maximize β since a higher β involves fewer datastore accesses. The minimum possible α for any key is 0 since a key can be written in one access and read in the next. There is no true upper bound on β since an extremely frequently-used key may be maintained at the proxy indefinitely (which we will discuss later). Notably, maintaining an α of 0 for all keys is impractical since any keys that are written to the datastore must be read (and deleted) in the following access. Assuming we maintain a number of keys less than the total possible amount at the proxy, we will need to write keys to the datastore, and reading all written keys from the previous access is infeasible. In this project, we search for ways to obtain a low α and high β based on a workload RideGuard may face. As a benchmark, we use the medium security guarantees presented in the Waffle paper, namely α=972 and β=5.

To achieve the second property, Waffle maintains a timestamp that is incremented every time the protocol is run. When Waffle receives a plaintext key and wants to compute the corresponding datastore key, it hashes the plaintext key with the timestamp (while maintaining a mapping between plaintext and datastore keys). Hence, we never write the same datastore key twice, even if they refer to the same plaintext key, since the timestamps will differ.

3.3 Waffle Components

As mentioned earlier, Waffle's privacy model involves holding keys at the proxy. To this end, the Waffle proxy maintains an LRU cache of size C, initialized with randomly chosen keys. Additionally, Waffle processes requests in batches of R plaintext keys, which may contain duplicates (we refer to the number of unique, non-cached keys as r). If a request contains keys in the cache, those keys are served directly from the cache. To ensure that all possible non-cached keys are written to the datastore, Waffle sends batches of B keys to the datastore at each run of the protocol, where the batch consists of r real queries and f_R fake queries on non-cached real keys. To determine which keys to send, the Waffle proxy maintains a balanced binary search tree containing the latest access timestamp for each key, and uses the f_R non-cached keys with the minimum timestamps from the BST. Optionally, the same procedure can be done with D dummy keys known in advance (always non-cached) and f_D fake queries on dummy keys sent in each batch. The constraint B=r+f_R+f_D ensures that the same batch size is sent to the datastore each time the Waffle protocol is run. Hence, the bounds in 3.1 and 3.2 for α and β were derived in the Waffle paper.

$$\alpha = \left\lceil \max\left(\frac{(N-C)-(B-f_D)}{B-R-f_D}, \frac{D}{f_D} \right) \right\rceil \tag{3.1}$$

$$\beta = \left\lfloor \frac{C}{B-fD+R} - 1 \right\rfloor \tag{3.2}$$

The Waffle protocol operates in two phases. In the first phase, a "read batch" of B keys is assembled as shown above and read from the datastore (this is done for both read and write queries, since we want to delete an out-of-date value at the datastore; for writes, the new value is put in the cache, meaning it can temporarily become as large as C+R). Since each read batch contains r real queries for non-cached real keys, fR fake queries on non-cached real keys, and fD fake queries on dummy keys, all keys in the batch are either non-cached real keys or dummy keys. This batch is sent to the datastore to be read. After receiving the read batch, a background thread issues requests to the datastore to delete the received keys at the datastore. In the second phase, for all non-cached real keys that are read, an element is evicted from the cache and added to a write batch, which is sent to the datastore. All other keys (guaranteed to be dummy keys) are added to the write batch with a randomly generated dummy value. As mentioned before, Waffle maintains a mapping between plaintext keys and datastore keys, and hashes keys with the timestamp before sending them to the datastore. To ensure no leakage of information from the values submitted to the datastore, Waffle encrypts them in an IND-CPA secure fashion under an encryption key stored at the proxy.

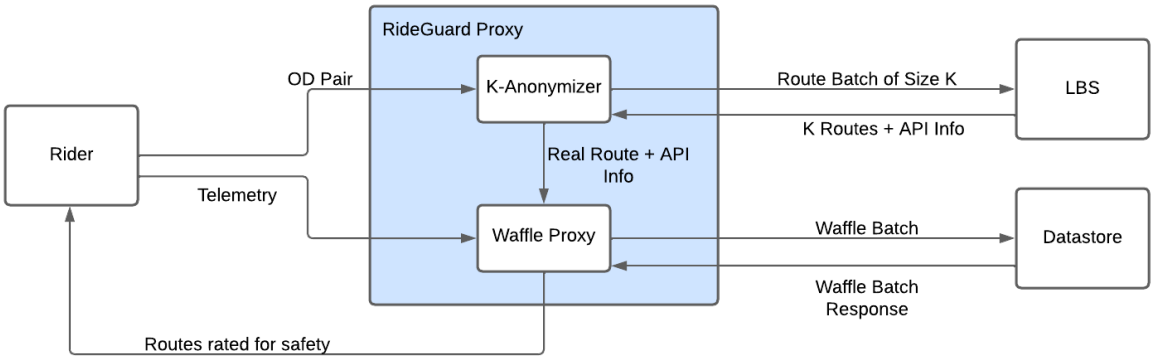

Figure 3.3: RideGuard System Diagram

3.4 Adapting RideGuard to Waffle

Our goal is to calculate safety indices for routes requested by riders while preserving α-β uniformity over datastore accesses, and not leaking any information while doing the calculations. In this section, we describe how to reconcile Waffle and the requirements of RideGuard with each other.

Safety Indices

Safety indices in RideGuard are composed of two main factors: information obtained from external APIs and telemetry submitted by riders, which can capture data such as events detected by sensors and rider experience level. This fusion of data sources allows RideGuard to effectively capture many aspects of rider safety. Hence, the end-to-end calculation of a safety index will consist of the following steps:

1. User submits route
2. RideGuard proxy generates K-1 fake OD pairs and geocodes all K OD pairs
3. K geocoded OD pairs submitted to a route generator, points along all K routes are sent to external APIs
4. Only for the real request, stored records are accessed via the Waffle protocol
5. Compute a safety index using the obtained records and API information for the real request

Most data collection for mobility applications is done through standards such as GBFS or MDS, where all records, regardless of the user, adopt a fixed schema. By contrast, RideGuard allows a safety index designer to specify custom schemas, allowing for service-area-specific indices to be designed. Additionally, the data for MDS and GBFS allow several records to be generated for the same location or time. Since our use case relies on computing statistics over records corresponding to a certain location and time, and Waffle requires accessed keys to be deleted after they are read once, a naive solution would involve deleting and writing many records for each run of the Waffle protocol. Hence, we require a more compact representation of the data needed to compute safety indices, and we leverage some domain-specific optimizations to make safety index calculation more manageable.

Since all safety index computations require a location and time, we can discretize location-time pairs using geohashes and the hour of the timestamp of the query. We can use the geohash and time to make storage keys for telemetry records (spatio-temporal keys). We can augment the spatio-temporal keys with information such as vehicle type to support more detailed queries for safety information. In keeping with the Waffle protocol, information about location and time is combined with access timestamps to create access keys. Additionally, we assume that all requests to calculate a safety index will only require aggregation functions (e.g. mean, standard deviation, count), so the final safety index involves combining several aggregation outputs.

Avoiding Leakages

Notably, one of our goals is to avoid leaking any additional information when computing safety indices. To achieve this goal, all stored values need to be of a fixed size. One way of

achieving this uniformity is to have each key-value pair correspond to a spatio-temporal key and a mapping between string keys and values of predetermined statistics needed to calculate a safety index (the stored value). This causes the stored record size to be independent of the number of submitted telemetry records. For example, calculating the mean rider experience level requires two values: a count of riders and a sum of experience levels, regardless of how often the spatio-temporal key is sent a telemetry record. To support a fixed amount of storage, statistics in RideGuard must be calculated in an online fashion, which is attainable for several popular aggregations.

A consequence of the uniformity of records in RideGuard is that detecting sensor events must be done on the edge. Detecting events from sensor data involves storing a variable number of datapoints for spatio-temporal keys. Fortunately, there has been significant work in detecting such events on the edge, using accelerometers and gyroscopes built into phones. Many routing apps are designed to work while mounted to the handlebars of a bicycle or scooter, and RideGuard is intended to leverage this behavior to gather information about rider behavior.

Since RideGuard needs to support calculating statistics in an online fashion, we need to modify Waffle to support two operations: reads (which are already supported) and updates. Waffle is designed to support reading and overwriting the value associated with a key. However, in RideGuard, we do not want to overwrite stored statistics, but rather update them based on their existing values. To this end, we never submit true writes to the Waffle proxy. Instead, when we want to calculate statistics from telemetry, we submit a read request and modify the data received from the datastore before inserting it in the proxy's cache. A consequence of this behavior is that all computation of safety indices is done at the RideGuard proxy, meaning no information is leaked while calculating the index since the proxy is trusted.

3.5 Waffle Implementation and Alpha Distribution

We implemented a Waffle proxy in Go, and primarily relied on a pool of 10 goroutines processing requests. Each thread accepts an OD pair, calculates K-1 fake OD pairs (which are subsequently reverse geocoded). In this phase, the proxy shuffles all K OD pairs and calls a route generator and other external APIs to obtain safety information. The thread then computes geohashes from only the real route and pushes them onto a FIFO queue, keeping geohashes from the same request in a single queue element. Our implementation used the Mapbox API, which provides a list of coordinates along a particular route, giving us a basis to calculate geohashes along the route. If there are enough geohashes in the queue, the thread runs the Waffle protocol on as many requests on the start of the queue as possible so as to not exceed R geohashes (pushing and popping from the queue atomically). Having fewer than R geohashes is effectively the same as duplicating requests in Waffle, so the protocol covers this case. Specifically, this is accounted for when deciding fR, since we use the number of unique keys r instead of the total number of input requests R to decide how many new requests

to use to get B total. The proxy also maintains a thread that pushes telemetry requests onto the queue atomically, and these telemetry requests are counted for running the Waffle protocol. Threads block until the corresponding run of the Waffle protocol is done. Notably, starvation is possible if the requests in the queue do not have enough geohashes (caused by not enough requests arriving). We have not implemented a fix for this, but it is easy to duplicate requests on the queue until there are enough to produce at least R geohashes. This can be done after some timeout of not executing a Waffle batch. Our experiments are hindered by Mapbox's rate-limiting, so we calculate latencies on a version of the proxy that uses results downloaded from the Mapbox API beforehand.

3.6 Choosing Waffle Parameters

We used a Flask server as the key-value store (and for calculating α values). We use geohashes of precision 7 (153 meters by 153 meters) to store information about fine-grained sections of the service area. There are 7106 such blocks in San Francisco, so we need to store N=170544 total keys (24 for each geohash). In choosing parameters, we first aimed for an α value exceeding the "medium security guarantee" presented in Waffle. Since RideGuard predominantly handles requests in real-time, we arrived at a cache size C=14212 (two hours of geohashes, 8.3% of data cached). To minimize storage and computational overhead at the RideGuard proxy, we set D (the total number of dummy keys), and f_D (the number of fake queries on dummy keys per batch) to 0. We use an even split between real queries and fake queries on real keys in Waffle batches. Choosing B=700, R=350, and f_R=350 achieves a theoretical maximum α of 445, exceeding the Waffle paper's value of 972 (lower is better). The theoretical minimum β for these parameters is 12, exceeding the Waffle paper's value of 5 (higher is better). Referring to Sybil's parameters, we set K=5 and consider the 5 most similar quadtree blocks.

To obtain a distribution of α values, we used trips gathered from a day of the BayWheels GBFS feed for January 2024 (1772 requests), ordered by timestamp and submitted by a pool of 20 threads. Since each request for a route can contain several geohashes, we make requests to the server with multiple geohashes at a time and wait until at least R geohashes are received to start the Waffle protocol (Waffle handles non-unique keys received by adaptively modifying the number of fake requests on real keys, preserving the proportions of requests to real and dummy keys given by B=R+f_R+f_D). We first requested to read all geohashes in a route at once (obtaining all safety information for a route), followed by sequentially sending requests to read (mimicking telemetry records) 8 times to each geohash.

3.7 Evaluation

We first examine the relationship between the distribution of alpha values and the skewed query pattern for locations in 3.5. Waffle offers a great deal of uniformity, with most key

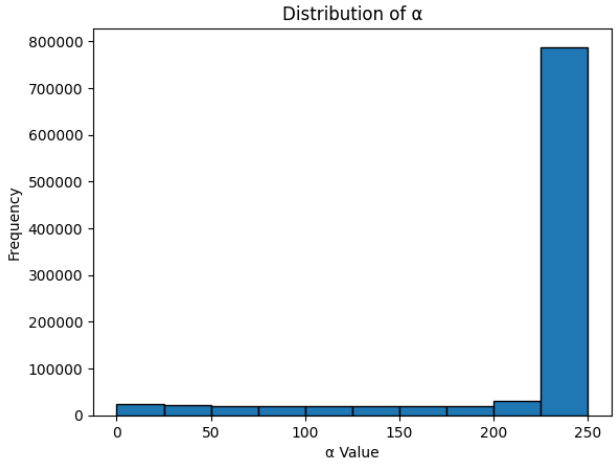

Figure 3.4: Distribution of α Values

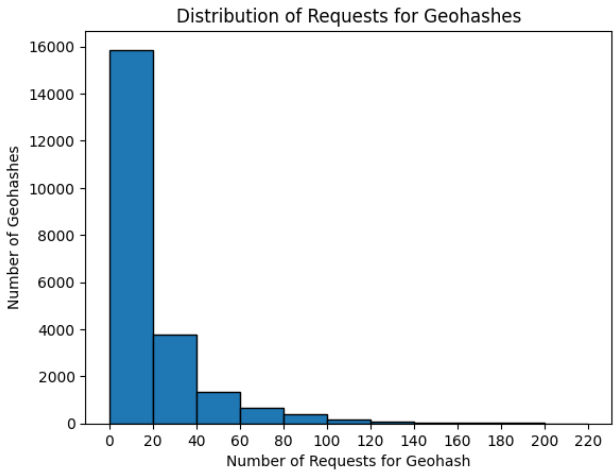

Figure 3.5: Number of Requests for Geohashes of Precision 7

accesses exhibiting an α between 225 and 250 in 3.4. The maximum observed α was 248, well below the theoretical maximum of 446. Notably, we still achieve a great deal of uniformity, with the majority of reads having an α of 225 or higher, despite the heavily skewed access pattern for geohashes. We are interested in the lower values of α to examine whether having a lower α correlates with usage. For this experiment only, we modify the getIndex function to attach the real key at the end of the hash. The key assignment is still non-static but allows us to examine the behavior of individual keys. We are looking for a wide variety of keys to be in each of the ranges outside of the 225-250 bucket, since this indicates that both keys corresponding to frequently accessed and non-frequently-accessed locations occur in the tail of the distribution. Table 3.1 shows that all parts of the distribution contain a variety of keys.

Since α-β uniformity does not provide complete uniformity over all accesses, we are interested in the extent to which geohashes with different amounts of accesses show up in the distribution of α values. 3.6 and 3.7 demonstrate that the geohashes corresponding to the most frequently requested spatio-temporal keys tend to have either the lowest or highest α values, surprisingly being far less frequent in the middle of the distribution.

Performance

We also examine the latency of route requests. Figure 3.8 demonstrates that the vast majority fall under 200 milliseconds. Additionally, we are interested in how the workload interacts with the Waffle proxy cache and request queue. To this end, we plot the number of requests taken in each Waffle batch and the number of geohashes taken in each Waffle batch. These metrics can capture how well the pool of 10 threads handles a mix of route requests and

Range	Count
0-25	23078
25-50	20708
50-75	19709
75-100	19649
100-125	19312
125-150	19287
150-175	19877
175-200	20832
200-225	41717
225-250	772288

Table 3.1: Counts of Keys in Each Bin of the α Distribution

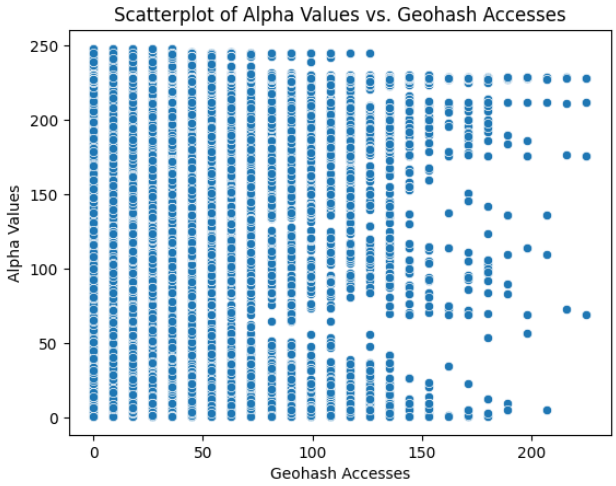

Figure 3.6: Distribution of α Values

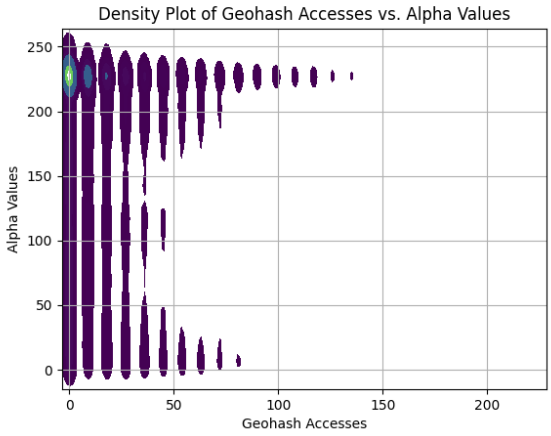

Figure 3.7: Number of Requests for Geohashes of Precision 7

telemetry requests. Figures 3.9 and 3.10 demonstrate that the number of geohashes taken is consistently close to the optimal value R=350, illustrating the efficiency of our proxy. Additionally, the number of requests taken per Waffle batch indicates a mix of telemetry and route requests being processed in each Waffle batch.

Figure 3.11 shows the hit rates over time for the Waffle batches, with an average of 0.9388, illustrating the importance of the Waffle cache in serving requests and obscuring access patterns. Notably, there are some batches with a lower hit rate, particularly at the start. This may be attributed to the initialization of the cache with random keys.

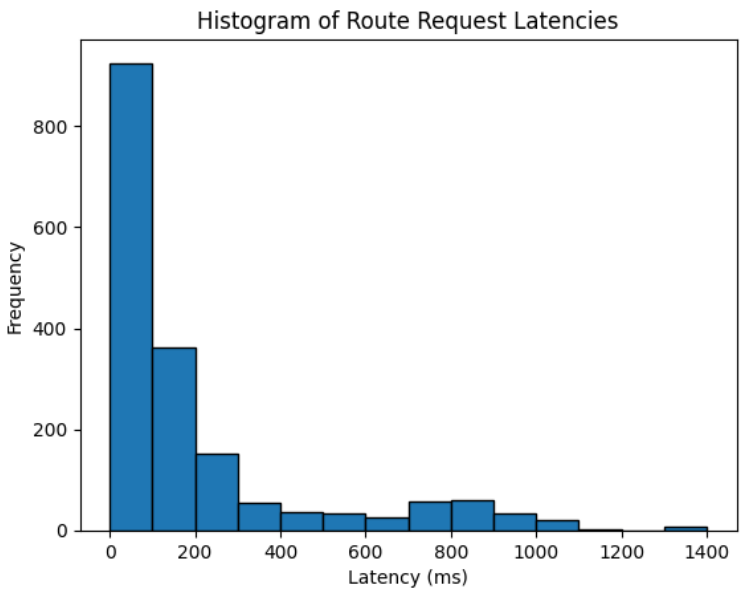

Figure 3.8: Histogram of Route Request Latencies

Variation of Cache Size

Since the proxy's cache plays an important role in serving RideGuard requests, we conducted tests to demonstrate the impact of cache size on results such as the distribution of α values, latencies, and Waffle proxy hit rates. We used three cache sizes: 3553 (half an hour of data), 28,424 (4 hours of data), and 56,848 (8 hours of data).

Figures 3.12, 3.13, and 3.14 demonstrate that while a higher cache size enhances obliviousness and lowers the observed values of α, they are associated with higher latencies for some requests, both overall and in Waffle batches, implying a higher bound on the worst-case execution time. Notably, the cases where 4 and 8 hours' worth of data were cached had some Waffle batches with a high hit rate and a high latency. This effect may be due to cache operations (in particular, those involving the LRU feature of the cache) taking longer due to a higher cache size. In the average case, however, there is a tradeoff between the maximum value of α that can be guaranteed and the amount of cache space at the proxy.

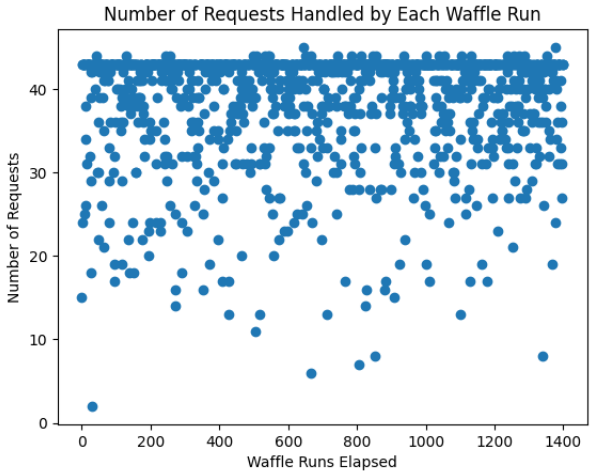

Figure 3.9: Number of Requests (Routes or Telemetry) Handled by Each Waffle Run

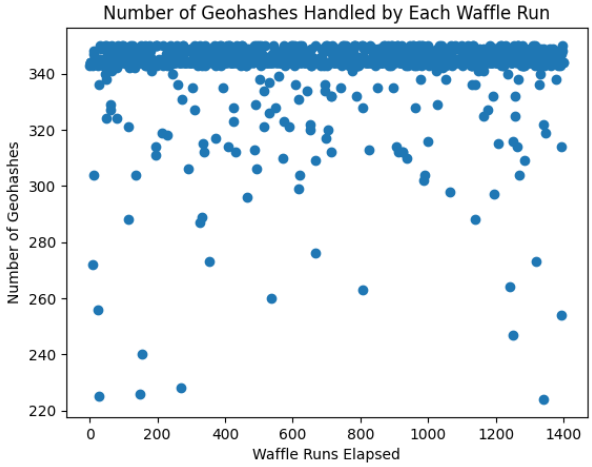

Figure 3.10: Number of Requests Handled by Each Waffle Run

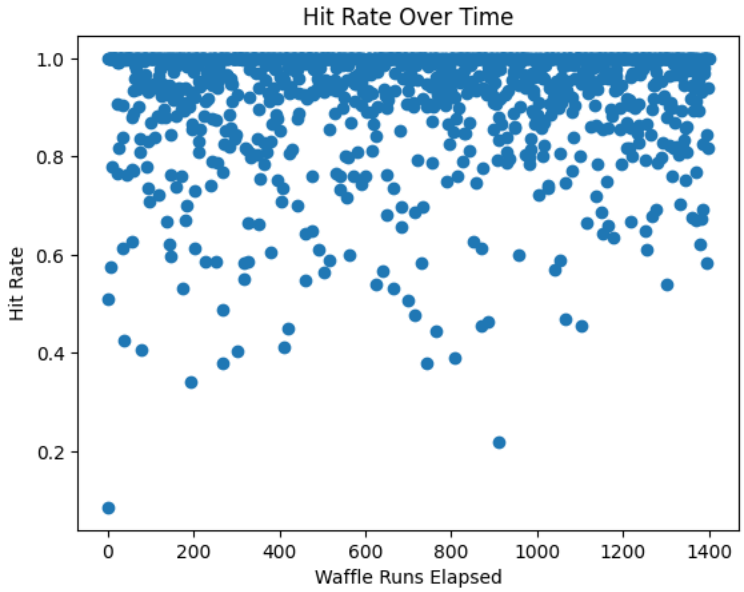

Figure 3.11: Waffle Batch Hit Rates Over Time

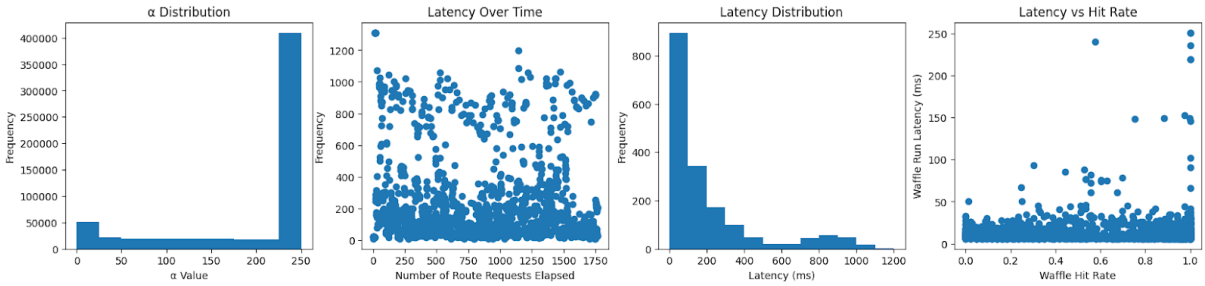

Figure 3.12: Performance Metrics Caching 0.5 Hours of Data

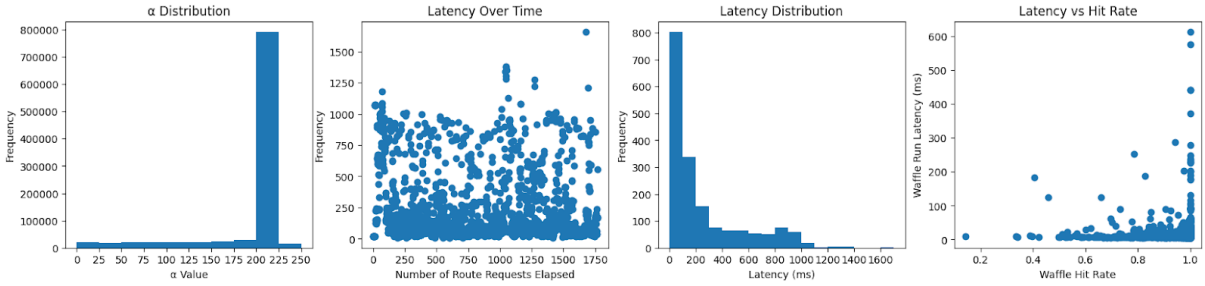

Figure 3.13: Performance Metrics Caching 4 Hours of Data

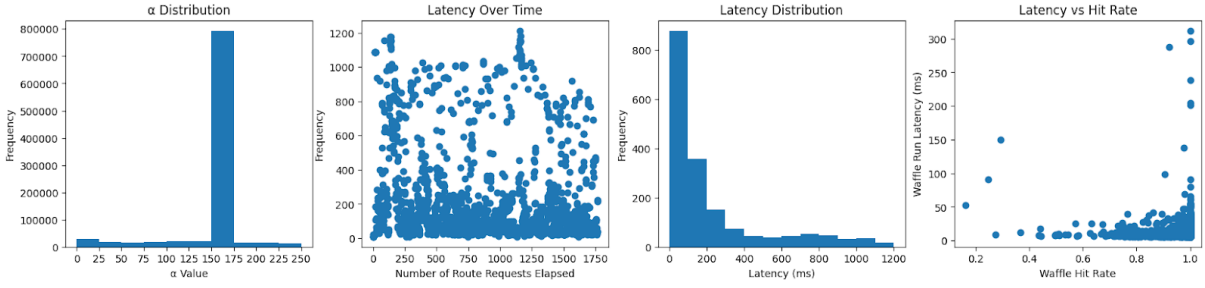

Figure 3.14: Performance Metrics Caching 8 Hours of Data

Chapter 4

Conclusion and Future Work

In this paper, we presented a method for calculating safety indices for micromobility while preserving rider privacy. We first devised a method based on SybilQuery that generates fake routes to maintain k-anonymity over location data when submitted to LBSs. We then presented a way to use the Waffle protocol to receive telemetry from riders while protecting against access pattern attacks from an untrusted datastore adversary. This enables the construction of safety indices for micromobility while preserving rider privacy.

4.1 Privacy Model for LBSs

Our privacy model for LBS data centered around k-anonymity, the practice of generating K-1 fake data points to send to an external source to obscure the real one. While conceptually simple, a major drawback of k-anonymity for API calls is that we need to send K times as much data as an insecure baseline. This can severely affect performance when calling several APIs to construct a safety index. A potential solution for this issue is to change the privacy model from k-anonymity to geo-indistinguishability, which entails perturbing the true OD pair, selecting points within some radius of the true origin and destination. The main pitfall is usability, since most micromobility trips are short, and a small radius can lead to easier reidentification. Additionally, the perturbed point should be trivially accessible from the original. However, this approach enables a single perturbed OD pair to be sent to LBSs.

4.2 Caching API Data

A number of different factors can affect micromobility safety, meaning that potentially several APIs need to be called while calculating a safety index. To reduce the overhead of calculating a safety index, we may be able to add a component that caches data from different APIs, indexed based on geohashes (and potentially other query parameters). The cached elements would be refreshed for a request requiring it, while the cached element is out-of-date.

4.3 Micromobility-Specific Data Collection

Our experiments were limited by the lack of available safety and telemetry data for micromobility vehicles. Firstly, we are not currently aware of any data sources that publicly release sensor data gathered from micromobility vehicles. Bicycles have been used to measure terrain quality, as in Gresenz, but different micromobility vehicles may exhibit significantly different dynamics, meaning that different sensor readings indicate safety-critical events such as swerves, crashes, and sudden stops. Chen devised D3, a machine learning-based approach to detecting safety-critical events, but their work centers on cars. Extending D3 to micromobility would enable the calculation of much more meaningful safety indices.

4.4 Customization

Our privacy model for queries relies on transition matrices between stations and geohashes, aggregated over the course of one year. A weakness of this approach is that it does not take factors like seasonality into account. For example, micromobility usage on the day of a popular event can influence the transit patterns of riders, and the k-anonymization must adjust to match. This could be fixed by implementing more fine-grained transition matrices (for example, weeks or days, as opposed to one year). Additionally, as micromobility usage grows, cities may choose to add new stations within service areas. To incorporate these new stations in the SS workflow without denying any rider requests, RideGuard needs to be able to predict the mobility patterns involving new stations, updating them as more riders use the station (and as data is released).

Bibliography

[1] 2018. URL: https://www.austintexas.gov/sites/default/files/files/Health/Epidemiology/APH_Dockless_Electric_Scooter_Study_5-2-19.pdf.

[2] URL: https://content.citymapper.com/news/2247/weve-partnered-with-lime-to-bring-you-more-active-travel-options.

[3] Chen Chen et al. *Micromobility Promises and Challenges in the Pacific Northwest*. 2022. URL: https://digital.lib.washington.edu/researchworks/bitstream/handle/1773/48584/Wang%20Micromobility%20Final-Report-Pactrans.pdf?sequence=1.

[4] Emma Dauterman et al. "DORY: An Encrypted Search System with Distributed Trust". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 1101–1119. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/dauterman-dory.

[5] Kevin Fang. "Micromobility injury events: Motor vehicle crashes and other transportation systems factors". In: *Transportation Research Interdisciplinary Perspectives* 14 (2022), p. 100574. ISSN: 2590-1982. DOI: https://doi.org/10.1016/j.trip.2022.100574. URL: https://www.sciencedirect.com/science/article/pii/S2590198222000379.

[6] Raphael Finkel and Jon Bentley. "Quad Trees: A Data Structure for Retrieval on Composite Keys." In: *Acta Inf.* 4 (Mar. 1974), pp. 1–9. DOI: 10.1007/BF00288933.

[7] Morteza Hossein Sabbaghian, David Llopis-Castelló, and Alfredo García. "A Safe Infrastructure for Micromobility: The Current State of Knowledge". In: *Sustainability* 15.13 (2023). ISSN: 2071-1050. DOI: 10.3390/su151310140. URL: https://www.mdpi.com/2071-1050/15/13/10140.

[8] Sujaya Maiyya et al. "Waffle: An Online Oblivious Datastore for Protecting Data Access Patterns". In: *Proc. ACM Manag. Data* 1.4 (Dec. 2023). DOI: 10.1145/3626760. URL: https://doi.org/10.1145/3626760.

[9] Simon Oya and Florian Kerschbaum. "Hiding the Access Pattern is Not Enough: Exploiting Search Pattern Leakage in Searchable Encryption". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 127–142. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/oya.

[10] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. "Arx: an encrypted database using semantically secure encryption". In: *Proc. VLDB Endow.* 12.11 (July 2019), pp. 1664–1678. ISSN: 2150-8097. DOI: 10.14778/3342263.3342641. URL: https://doi.org/10.14778/3342263.3342641.

[11] Raluca Ada Popa et al. "CryptDB: Processing Queries on an Encrypted Database". In: *Communications of The ACM - CACM* 55 (Sept. 2012). DOI: 10.1145/2330667.2330691.

[12] Grandview Research. 2022. URL: https://www.grandviewresearch.com/industry-analysis/electric-kick-scooters-market.

[13] Pravin Shankar, Vinod Ganapathy, and Liviu Iftode. "Privately querying location-based services with SybilQuery". In: *Proceedings of the 11th International Conference on Ubiquitous Computing*. UbiComp '09. Orlando, Florida, USA: Association for Computing Machinery, 2009, pp. 31–40. ISBN: 9781605584317. DOI: 10.1145/1620545.1620550. URL: https://doi.org/10.1145/1620545.1620550.

[14] James Tark. Sept. 2022. URL: https://www.cpsc.gov/s3fs-public/Micromobility-Products-Related-Deaths-Injuries-and-Hazard-Patterns-2017-2021.pdf.

[15] Peng Zhang et al. "ShiftRoute: Achieving Location Privacy for Map Services on Smartphones". In: *IEEE Transactions on Vehicular Technology* 67.5 (2018), pp. 4527–4538. DOI: 10.1109/TVT.2018.2791402.