Muddler: Using Oblivious RAM For A Privacy Preserving Location-Based Service

by

Danish Mehmood

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Science
in
Computer Science

Waterloo, Ontario, Canada, 2014

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

As smartphones become ever more prevalent, context aware applications are becoming increasingly popular. Location-based services such as Foursquare have been among the leaders of this trend. Some of the most popular location-based services offer users the ability to check-in to locations, leave tips for others and provide ratings. These applications require the user's location information to deliver a localized user experience. The release of this information raises some serious privacy concerns. We present Muddler, a privacy preserving location-based service modeled on Foursquare. The service is designed to be flexible and practical. It ensures user privacy, while withstanding threats that previously proposed designs have failed to address.

Muddler uses an Oblivious RAM based data storage that is manipulated by a secure coprocessor to ensure that adversaries cannot learn about user information even if they operate the service or simply observe traffic between entities in the system. The service also exposes a public API that provides venue owners with functionality that may help them understand user behavioral patterns in an attempt to make it commercially feasible. We describe our implementation in depth and explain how the API is implemented and also discuss possible use cases. We then present a performance analysis of Path ORAM, the Oblivious RAM scheme used. We explain how we simulated realistic user check-in distributions followed by an experimental evaluation of the system. The results validate the usefulness of our proposal.

Acknowledgements

I would like to thank Urs Hengartner, my supervisor for his guidance, support and patience. I would also like to thank my thesis readers, Douglas Stinson and Ian McKillop, for taking out the time from their busy schedules and providing me with valuable feedback. Finally, I would like to thank members of the Cryptography, Security, and Privacy research group at the University of Waterloo for making this an enjoyable experience.

Dedication

I dedicate this thesis to my parents, family, and friends.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Overview

The increasing adoption of smartphones has precipitated a paradigm shift in the world of personal computing. These mobile devices are overwhelmingly being used for tasks that were previously the domain of personal computers. The scale of this shift can be judged by the fact that more smartphones were sold globally as compared to 'dumb' feature phones in 2013 [60]. Over 1 billion smartphones were shipped in 2013 alone [54].

The current range of smartphones boast a number of sensors such as cameras, accelerometers, gyroscopes, GPS receivers, fingerprint scanners and even barometers and thermometers. The presence of these sensors coupled with increasingly powerful processors on board these devices have turned them into sophisticated computers that have ushered in a new era of intelligent applications. The observation of data coming in from the device's sensors enables smartphones to identify the context of the device owner and provide new feature rich applications and services.

The most common among these applications are location based services, making use of the user's location to provide filtered content based on the user's locality. This information may be determined with varying levels of precision from the GPS coordinates retrieved from the GPS receiver on board the smartphone, the Wi-Fi access points that the device is connected to or the phone's cellular network connection. For example, a service may use a client's location to provide localized weather updates, classifieds or infotainment services.

Coupling user location with their contacts can also be used to enable geo-social networking. This can involve providing proximity alerts triggered when friends are within a

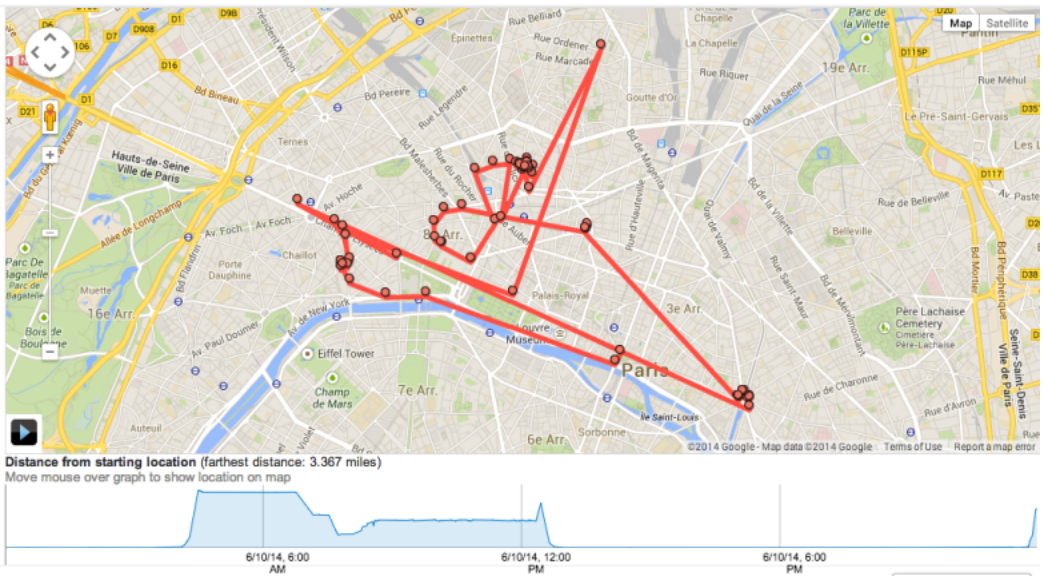


Figure 1.1: Google Maps location history [62]

certain neighborhood or sharing current location between contacts. Similar services may also enable the 'discovery' of like-minded people within a certain geographical area or provide location-based ratings. There are several scenarios where location-based services can prove to be crucial. For example in a disaster scenario these services can provide crucial on the ground information and help in managing the situation. Volunteers may work collaboratively to provide information. Another possible application can be a service where users report something as ordinary as traffic alerts.

The most popular real world manifestations of this concept include Foursquare [18], which is a search and discovery service providing users with a localized experience. Users can search for nearby locations, rate locations and leave tips. Yelp [66] is quite similar in nature. Foursquare currently boasts over 50 million users and also provides business managers access to data and analytics for their locations. Previously users could check-in at a geographical location on Foursquare and share it with their followers. This feature has now been transferred to a new service, which does not even require users to check-in to share their location. Instead it introduced neighborhood sharing, which lets friends on the network know of the neighborhood the user is currently in. Since these services require periodic access to a user's location it raises a very serious concern, that of a user's location privacy.

1.2 Location Privacy

Most location-based services require the periodic release of a user's location to a central server. Depending upon the frequency of this release a server may be able to infer a lot more than just where the user has been most recently. For example, collecting this information at intervals over a long period of time provides the server with an entire history of a user's mobility recorded to increasing levels of geo-spatial precision. Figure 1.1 shows a user's location history as recorded by Google Maps [43]. This information when coupled with identifying information that a user may have provided to the service only increases the possibility of a breach of a user's privacy revealing more about a user's lifestyle than they could have imagined when signing up. Even if a user has not provided such information, other sources of information in the public domain such as census data, the yellow pages or even the telephone directory can be used to identify the locations that a user frequents. An analysis of these location traces, their times of origin and these other information sources could possibly even help in the identification of a user's home address [35] or even reveal visits to sensitive locations such as an abortion clinic.

The sensitivity of this information can be judged by the fact that even publicly available information such as tips are enough to enable adversaries to identify a user's home city [53]. Simply put, if a user publicly shares their current location they are basically advertising where they are not [6]. While sharing information definitely has its risks, not sharing at all is simply not the best solution. This is because location based services do have the potential to contribute to the well-being of society by enabling activities such as urban activity inference [45] which may help town planning and administration and recommender systems [46, 40]. In most cases, users would like to have some degree of control over this information. Hence, location privacy [10] is defined as the right of the user to control access to this information.

Existing research in location privacy has several proposals based on anonymizing [25, 58, 34] identifiable user information so that some semblance of privacy can be maintained. Obfuscation and cloaking techniques [39, 67, 28] propose adding dummy trajectories or releasing imprecise locations. However, most of these proposals fail to preserve a user's privacy if user behavior is observed over a long period of time, while the rest are application specific like most cryptographic protocols [68, 44] and have limited use cases. Private Information Retrieval [33, 47] provides promising results but these schemes are inherently unsuitable for frequent location updates and are computationally intensive. Some existing research has followed TaintDroid's [15] taint propagation approach to track private information as it flows within the smartphone and blocking access to the information altogether or faking it [9, 29] but this necessarily results in degradation of service or unwanted side-

effects. In general, research in this area has focused on releasing information in a privacy preserving manner. Such a solution does not fare well when the goal is to store information so that it can also be retrieved later.

Several privacy preserving location sharing frameworks have also been proposed [32, 26, 17] but they either require faith in non-colluding entities or are susceptible to traffic analysis attacks. Most recently Zerosquare [50] proposed separating user profile data and user location data. Although this is an interesting idea, the framework also suffers from susceptibilities faced by other such frameworks. Zerosquare's threat model does not even include tracking attacks, while most of the other solutions are not well suited to storing information such that it can later be retrieved and updated. Oblivious RAM [21, 22] schemes enable accessing items in memory while hiding access patterns. We argue that Oblivious RAM schemes, when combined with a secure coprocessor [30], are ideally suited for such a purpose.

1.3 Goals

The goal of this work is to present the design of a privacy preserving location-based service modeled on Foursquare. In this section we present the goals of our design:

- **Functionality:** We want the service to allow users to check-in at locations and leave tips if they so desire. The service should also support the provision of business analytics to venue owners to make the design commercially viable.
- **Scalable privacy:** The design should be suitable for something as widely used as Foursquare and a user's location privacy should not be violated even if the server storing location check-ins is observing incoming check-ins. In particular the service provider should be unable to identify the location that a check-in is associated with. The service should also be able to withstand tracking attacks. This means that adversaries should be unable to figure out the geospatial proximity between two location check-ins even if they were created at the same location.
- **Flexible:** Although we are presenting the design of a very specific service, our goal is to make the system flexible enough to be easily extensible and applicable to a wide array of services and not just location check-ins. Our goal is to design a system that others might find relevant when they look at developing similar services.

- **Computational economy:** Since we are designing a location-based service specifically for smartphone users, our goal is to make sure that client side computation is minimized as these devices have limited power availability. Our aim is to ensure that our system does not impose computational requirements that drain out the battery and inconvenience the user. We want a system that provides an easily adoptable solution to casual users instead of offering a stark trade-off between privacy and usability.

1.4 Our Contribution

Since Foursquare is one of the most popular applications in its category and provides an essential service, that of rating locations and providing tips, we propose a location-based service in a similar vein. We present Muddler, a privacy preserving location based service modeled on Foursquare. The goal of this project is to propose a commercially viable service similar to Foursquare and Yelp, while ensuring the privacy of a user's location. This work builds upon the location-indexed database described in Zerosquare and presents a privacy-preserving location-based service modeled on Foursquare's check-in functionality coupled with its business services. The business services offered by Muddler include providing venue owners with statistical information such as reports on the total number of check-ins, the number of unique visitors and visitor frequency. Unlike Zerosquare however, our proposal is not vulnerable to tracking attacks and does not rely on multiple non-colluding entities.

While designing Muddler, we aimed at making the design flexible enough to be tweaked into provisioning for other location-based services in the future. Our design consists of an ORAM based location-indexed database. This database stores all user check-ins and any associated information such as tips. It is manipulated by a secure coprocessor, which can be thought of as a black box. This essentially turns the ORAM into a Private Information Retrieval scheme but with the benefit of being able to add updates to the database. All communication between the secure coprocessor and the database takes place in an encrypted fashion. User devices communicate directly with the secure coprocessor. All such communication is encrypted with the secure coprocessor's public key. The secure coprocessor exposes an API, which is used to perform store and retrieve operations. We make the following contributions:

- We present the design of Muddler, a privacy preserving location-based service that is modeled on a real-world location-based service but which uses an ORAM based server storage. The design incorporates services inspired by Foursquare's business services to make the design commercially viable.

- We implement the complete design and provide experimental results of the performance of our proposed service.
- We generate a realistic check-in data set and then test the performance of our proposed service under realistic workloads.

The rest of this thesis is organized as follows. Chapter 2 discusses the related work in this area while chapter 3 presents Path ORAM. In chapter 4 we present the design and architecture of our system, the implementation of which is described in chapter 5. We provide a performance analysis and results of our work in chapter 6. Possible future work is discussed in chapter 7, while we conclude in chapter 8.

Chapter 2

Related Work

This section presents the related work in this area. The related research to this work can be classified into two broad categories. Section 2.1 focuses on prior work in the area of location privacy and discusses how that is related to our work. Section 2.2 on the other hand presents the evolution of ORAM schemes and discusses the trade-offs between different schemes.

2.1 Location Privacy for Location-Based Services

Several approaches have been proposed to address location privacy issues in location-based services ranging from anonymization to generalization or perturbation. While anonymization techniques for location privacy fail to live up to the promise [58] when an attacker has auxiliary information on the victims, the generalization or perturbation of location traces requires the addition of too much noise [35] making most location based applications impractical. Specialized protocols or cryptographic techniques provide privacy preserving alternatives but they are usually very application specific and can't be generalized to applications with different use cases. The following subsections present a brief summary of the more popular approaches. This is by no means an exhaustive review. The reader may find the brief surveys by Krumm [36] and Terrovitis [61] pertinent.

2.1.1 Anonymization

A lot of the preliminary research in location privacy was carried out in this direction. The idea of a long-term pseudonym, an alias that hides the user's identity, does not guarantee privacy as shown by Beresford et al. [10] who were correctly able to de-anonymize all users participating in their study. As a consequence, the idea of mix zones was proposed, where users frequently changed their pseudonyms with other users in the zone. The fact that this approach was unable to provide user's adequate location privacy in the office environment where the experiments were conducted shows the limitations of this approach.

Gruteser and Grunwald [25] proposed using a central location server which receives very high precision position information from clients. This central anonymizer communicates with external services on behalf of the client after stripping identifiable information from the received encrypted messages and perturbing the revealed location so that a required level of anonymity is maintained. K-anonymity [55] with respect to a user's location is described as the condition of a user's location being indistinct from $k-1$ other users. This necessarily precipitates the generalization of a user's location information to the extent that it is the same as $k-1$ other users possibly becoming utterly useless for several precision sensitive applications. Interestingly enough, this approach is not suitable for location-based services. Consider the case where the required k users are all located in a very small space [58], the uncertainty about user locations will be very low. Another problem with this approach is that the location anonymizer becomes a central point of failure.

A very interesting approach proposed by Kido et al. [34] requires the generation of several false or dummy position data requests with every true request to make real requests indistinguishable from false ones for the service provider. This necessarily increases the communication overhead and possibly the computational overhead depending on the application. The authors do describe some cost reduction techniques but they are very application specific and will not generalize.

2.1.2 Spatial and Temporal Cloaking

The strength of location privacy in this context can be defined as the extent of the ambiguity in the precision of the location. Leonhardt et al. [39] present a three layer access control mechanism. The first layer specifies the authorization requirements of the service to be able to perform a successful query. The second layer specifies the precision of the location information to be released, while the third layer specifies the amount of detail to be released about identity information. Each of these layers is defined as a set of policies. This enables

the substitution of precise location information with less detailed information as objects are organized in a hierarchy. For example, a staircase might be at the bottom of this hierarchy while building might be at a higher level. Thus, the information released will most likely lack precision.

Tun et al. [67] present the idea of introducing dummy trajectories that intersect human trajectories confusing observers. This scheme is vulnerable to being exposed if the adversary observes the long-term behavior of the users. Tracking a user can result in the adversary linking information about other places to the user's identity. Reducing the location sampling frequency [27] decreases the success rate of this attack. Hoh et al. [28] further go on to describe a time-to-confusion metric, which evaluates location privacy. This metric defines the duration that an adversary can track a user without being confused. The proposed algorithm guarantees a maximum time to confusion by only releasing location data such that the level of confusion specified is maintained. This necessarily implies that all location updates will not be released and may have an impact on the quality of the service provided.

2.1.3 Cryptographic Techniques

While the techniques discussed above are general and can be deployed in a wide range of applications, this subsection of this chapter describes protocols that are application specific and generally fall in the domain of secure multi-party computation, where multiple participants collaborate in computing the result of a function without revealing their inputs to each other or any third party. For example, participants may work together to ensure that a user's location is only shared with a friend if the friend is in close proximity [68]. Similar techniques may also be used to share a user's location with friends. In such a scenario, the inputs are typically encrypted and the cipher text is manipulated using cryptographic techniques. For example, Homomorphic [49] or Commutative Encryption [52] may be used to perform computations over the cipher text without revealing the plain text to achieve the same result in encrypted form. An interesting idea here is the extraction of participating individuals extracting location tags [44]. Subsequently the participants can use private set intersection using a homomorphic encryption scheme to determine if the intersection of their sets of location tags exceeds a certain threshold. The Shy Mayor [11] proposes cryptographic protocols that enable the verification of user check-in and co-location claims. Although it would fall in this category, the protocols proposed apply to very specific use cases.

Another approach that falls under this range of solutions is Private Information Retrieval (PIR) [12], which allows clients to query databases without leaking information

about the records accessed to the host server. PIR solutions in the context of location privacy [33, 47] generally involve combining cloaking techniques with privacy preserving querying. On a high level this involves creating regions around the user and performing PIR queries within those regions. Private Information Retrieval schemes are not well suited for frequent updates and are generally computationally intensive.

2.1.4 Privacy Preserving Location Sharing Frameworks

Some recent work has focused on proposing general frameworks for privacy preserving location-based services. Trust No One [32] proposes a system where no one entity has information on both user queries and user location. This information is distributed such that the service provider only has information on a user's queries and a third party, most likely the mobile service provider has a user's location information. This is an attempt at ensuring that one entity should not be able to accumulate all of a user's private information. The mobile service provider assigns pseudonymized location values to the user's current location, while the service provider assigns pseudonymized identifiers to all the entities that have subscribed to the service. A decentralized matching service is used to exchange information between entities in the system, to ensure that no attacks by malicious users with actual information on any user in an attempt to identify the other pieces of information can take place. This system is susceptible to collusion attacks, from mobile users who contribute the proxies in the matching service.

Koi [26] works along similar lines, using privacy-preserving location-based matching to avoid exposing fine-grained location information to applications on the phone. It consists of a mobile and a cloud component. The mobile component liaises with applications and the cloud exposes an API that allows the registration of clients and triggers, which act as queries providing notifications through call backs when a match is found. The cloud component consists of a non-colluding matcher and a combiner. The matcher is aware of a user's identity and attributes including location without knowledge of the association between them. The combiner on the other hand is aware of the association between users and their attributes without knowledge of their actual values. Although there is a privacy-preserving protocol that enables these two cloud components to perform matching without learning the association between users and their attributes, this system is susceptible to traffic- analysis attacks when updates are observed over time.

Recent work includes The Location Privacy Guardian [17], which presents a client-side solution and Zerosquare [50]. The Location Privacy Guardian provides privacy guarantees on a per-app basis. It consists of a location interceptor, which intercepts all incoming

location requests from apps and forwards them to a location anonymizer. A rule manager determines the anonymization strategy for the anonymizer after interacting with the user. The interceptor then provides the anonymized location to the requesting app. The Location Privacy Guardian provides low privacy guarantees for users living in sparsely populated areas as the protection mechanism uses a model relying on census data. Furthermore, it requires modifying the mobile platform it is deployed on. Zerosquare on the other hand proposes a unique solution. It proposes separating user profile data from user location data. The goal is to ensure that no one entity should be able to obtain both. Zerosquare consists of a user-indexed database that stores user information, a non-colluding location-indexed database that stores information on locations and optional cloud components. Although this does provide a flexible framework, that has some serious loopholes. It is vulnerable to tracking attacks. The problem with this approach is that the location database can observe the sequence of check-ins and observe the spatial and temporal proximity between them. This information when recorded over a long time period may be used to guess possible user trajectories. If these trajectories are guessed over a long period of time common patterns will have an even higher probability of depicting the reality. If sensitive identifiable locations are included in that trajectory such as a user's home or work place or unique locations that a user frequents during certain times of the day, identifying a user may be possible resulting in a privacy breach.

2.1.5 Summary

For the specific use case that we are interested in, being able to update the database with frequent additions is an important requirement. Furthermore, any solution that we propose should be able to ensure user privacy under most circumstances. Previous work in this area has proposed several solutions but they are just not suitable enough for something that is as scalable and as widely used as Foursquare. They are either fundamentally flawed allowing certain kinds of attacks or they are just not designed to support the massive number of updates and retrievals such a system would require. While some of the more secure approaches have very specific applications and can not be generalized to other use cases, others are either too computationally expensive or are vulnerable to tracking or traffic analysis attacks. As a consequence there rises a need for a scheme that is robust and computationally feasible. We explore using an ORAM based data storage for the purpose of our location-based service and analyze how it fares against these functional requirements.

2.2 Oblivious RAM

The observation of the execution of a piece of software can enable an adversary to identify the essential characteristics of that program. The adversary can obtain this information by monitoring the memory locations accessed during execution, which may leak information about various programming constructs such as loops. Encrypting the contents of the computer's memory does not solve this problem as the adversary can still observe the memory locations accessed and the frequency of access. Recent work by Islam et al. [31] shows that observing data access patterns of an encrypted email dataset using a heuristic solution can successfully identify 80% of the queries.

The standard security definition for ORAM provides the following guarantees:

1. The adversary should not be able to identify what is being accessed.
2. The adversary should not be able to estimate the age of the data being accessed or the time since it was last accessed.
3. The adversary should not be able to identify any similarities in the access pattern.
4. The adversary should not be able to detect if the access is a read or a write.

2.2.1 Hierarchical Solution

Goldreich and Ostrovsky [21, 22, 48] originally proposed Oblivious RAM (ORAM) to hide the execution of software from being observed. ORAM schemes continuously shuffle and re-encrypt data as it is being accessed. This ensures that the adversary has no idea as to the data that is being accessed even if the memory location is observed and thus prevents software reverse engineering. The same idea can be extended to the cloud computing scenario where it enables a client to completely obfuscate the underlying data access patterns from the untrusted remote server.

The seminal work by Goldreich and Ostrovsky [22] presented the *Hierarchical Solution*, which arranges the ORAM in a hierarchy of levels with increasingly more capacity for elements. Elements are assigned to buckets on each level using a hash function. When an element is accessed the algorithm accesses a bucket on each level. The bucket to be accessed is selected based on the hash function. If the element being looked for does not exist on the current level, a dummy item is accessed. Once the item is found it is updated and re-inserted at the top of the hierarchy. If a level is full then the elements it contains are

obliviously rehashed into the next level. The use of expensive oblivious sorting to reshuffle the ORAM periodically effectively rendered this an impractical solution with an amortized access overhead of $O((\log N)^3)$ with an unacceptably high constant. Several subsequent works aimed at optimizing and improving upon this *Hierarchical Solution*.

Deploying the client on a cryptographic secure coprocessor [30] effectively creates an ORAM based *Private Information Retrieval* protocol [64]. Williams et al. [65] later extend this work, which provides no correctness guarantees using an encrypted bloom filter and an encrypted hashtable to store each level of the hierarchy. This allows the efficient identification of the level where the searched element lies.This scheme reduces the required server side storage from $O(N \log N)$ to $O(N)$. Using cuckoo hashing [24, 51] and newer oblivious sorting algorithms [23] imposes similar external storage requirements. Interestingly enough, Kushilevitz et al. [37] show that the scheme described in [51] is not secure claiming that cuckoo hashes may possibly lead to insecure schemes. On the other hand they also show that [65] using Bloom filters results in a scheme with a non-negligible chance of being broken.

2.2.2 Binary Tree Framework

Recent advancements include the Binary Tree framework proposed by Shi et al. [57]. The data storage is organized as a binary tree. Each node in this tree is called a *bucket* and has a fixed capacity of elements, each of which is encrypted using an authenticated encryption scheme to disallow any tampering by adversaries. Every element that is added to the tree is randomly assigned to one of the leaf buckets. This mapping of elements to leaf buckets is stored in a data structure on the client, which can be referred to as a *position map*. Elements being written to the ORAM are always added to the root bucket. If a read request for an element is made, all the buckets that lie along the path from the root bucket to the assigned leaf bucket are accessed. Each bucket iterates over all the elements it contains decrypting them. If a match is found it is removed and replaced with an encrypted dummy element, otherwise the elements are randomly re-encrypted. The removed element is then returned. If the element is not found a dummy element is returned to ensure obliviousness. Every time an element is removed, it is randomly assigned to a new leaf bucket, the mapping is updated in the position map and the element is inserted into the root. Since, each bucket has a fixed capacity there is the possibility of overflow. To prevent this, a periodic eviction of elements from non-leaf buckets takes place in the background. A specified number of buckets are randomly selected at every level of the tree for eviction. An element is popped from each of these bucket ORAMs and writes are performed on both the children of the bucket to ensure obliviousness. A real block will be written to the child

that lies along the path from the leaf bucket to the root, while the other child will receive a dummy write. If the popped element is a dummy then dummy writes will take place on both the children. This scheme significantly simplifies ORAM implementations and enjoys much simpler proofs of security.

Obliviad [8] presents an ORAM based privacy preserving architecture for online behavioral advertising based on this Binary Tree framework using a similar construction to the one described above. The goal is to prevent the leakage of any user profile information. For this purpose, Obliviad deploys a secure coprocessor to implement an ORAM based Private Information Retrieval scheme. Ads are stored against keywords used to describe user behavioral profiles. Every element in the bucket consists of a keyword and an ad. The mapping of keywords to leaf buckets is stored on the secure coprocessor. This construction allows the insertion of multiple elements with the same keyword and distinct advertising payloads into the path.The subsequent write back operation adds all these elements back to the root node. The maximum number of elements for a certain keyword is constrained by the maximum bucket size. A background eviction process takes place to prevent the possibility of overflow. In essence this work happens to be the closest to ours.

A subsequent proposal by Chung et al. [14] provides statistical security but also has an $O((\log N)^3)$ overhead. Path ORAM proposed by Stefanov et al. [59] has an $O((\log N)^2)$ worst-case cost and introduces a novel bottom up eviction scheme. It also introduces a data storage called the stash, which can be thought of as the working memory on the client. When an element is being read, the path that it is assigned to is read into the stash in its entirety. While writing a path back to the tree any possible elements in the stash that lie along that path are evicted into the tree. At non-leaf levels elements assigned to other paths that intersect that intermediate node can be also evicted. This ensures that the tree is heavier towards the bottom and discourages the need for a background eviction subroutine that is invoked independently. This also results in a much simpler implementation.

Chung et al. [13] simplify the implementation of the stash with a standard HashTable and a Queue. Although, this greatly simplifies implementation it does introduce some extra overheads resulting in a worst-case overhead that is worse than Path ORAM.

The efficiency of an ORAM scheme is judged on three main characteristics:

- The amount of client storage required.
- The amount of remote server storage required.
- The overhead of accessing data.

Algorithm	Amortized Cost	Client storage	Server storage
Shi et al. [57]	$O((\log N)^3)$	$O(1)$	$O(NlogN)$
Goldreich et al. [22]	$O((\log N)^3)$	$O(1)$	$O(NlogN)$
Williams et al. [65]	$O(\log N \log\log N)$	$O(\sqrt{N})$	$O(N)$
Pinkas et al. [51]	$O((\log N)^2)$	$O(1)$	$O(N)$
Williams et al. [64]	$O((\log N)^2)$	$O(\sqrt{N})$	$O(N \log N)$
Stefanov et al. [59]	$O((\log N)^2)$	$O(\log N)$	$O(N)$

Table 2.1: A comparison of various ORAM schemes.

Table 2.1 presents these characteristics of some of the algorithms that have been discussed. Compared to other schemes, Path ORAM requires a very simple implementation, has statistics comparable with some of the best schemes with limited client storage and is practically efficient. For this reason, we choose Path ORAM for the purpose of Muddler.

Chapter 3

Path ORAM

This chapter provides a brief explanation of Path ORAM in Section 3.1. It then discusses how our construction builds upon Path ORAM to ensure that this algorithm can be used for the purposes of a location-based service in Section 3.2.

3.1 Overview

Path ORAM [59] builds upon the binary tree framework proposed by Shi et al. [57]. The objective is to prevent any information leakage to adversaries observing data access patterns. Since encryption alone does not provide this guarantee, Path ORAM shuffles data and randomly re-encrypts it as it is being accessed. For this purpose, the database is stored on the untrusted server, which may be remote is organized as a binary tree. A binary tree is used only for the sake of simplicity. Data is stored in atomic units called *blocks*. Each of these blocks has a unique block identifier *id* and a data payload. Every node of the tree is called a bucket and can contain a maximum number of m blocks. Each of these blocks is assigned to a leaf bucket selected uniformly at random from all the leafs. The client stores data structures called a *position map* and a *stash*. The position map stores the assignment of blocks to leaf buckets. It contains a mapping of block ids against leaf bucket ids. The size of this data structure increases or decreases depending upon the number of blocks in the tree. The stash on the other hand can be thought of as the client's local working memory.

Initially, the tree is filled with dummy blocks. At any point in time a block can lie either in the stash on the client or on the server along the *path* from the root to

the assigned leaf bucket. The client manipulates the ORAM through the ReadBucket() and WriteBucket(*elements*) operations that are supported by every bucket. The ReadBucket operation decrypts all the blocks in the bucket and returns them, while the WriteBucket(*elements*) operation overwrites the blocks in the bucket with *elements* and randomly encrypts them. If the bucket is not full then it is padded with dummy blocks. An encryption scheme that provides ciphertext indistinguishability is used.

When a block is requested, the client looks up the block's id in the position map to identify the assigned leaf bucket. The entire path from the root to the assigned leaf bucket is read into the stash. The requested block can then be returned. Dummy blocks are ignored during this process and are not added into the stash. If a write is intended then the block is updated in the stash. The block is assigned to a new leaf bucket selected at random. This random re-assignment of blocks ensures that the adversary is unable to distinguish between requests for the same block or distinct blocks. Every time a block is re-assigned the assignment stored in the position map is updated. If the block does not exist then a random path is read into the stash instead to preserve obliviousness. The path is then written back to the server.

Path ORAM strives to push blocks as deep down in the tree as possible. The goal is to try to empty the stash as much as possible. The write back starts from the leaf nodes. Any blocks that are assigned to the leaf bucket of the path being written back are evicted first. Since a block can lie anywhere along a path from the root to its assigned leaf bucket, the path being written back can have other such paths ending in different leaf buckets intersect its intermediate buckets. Blocks assigned to these other paths are also evicted into the buckets at the appropriate levels. Any leftover blocks remain in the stash and can be written back in subsequent accesses. Because of this the stash has a very high probability of being empty after reads. Algorithm 1 presents the pseudo-code for Path ORAM.

Figure 3.1 shows a sample execution pattern for one path ORAM access where the block with block_id b3 is requested. Figure 3.1a shows the state of the server and client storages before the block is requested. The stash only contains block b2, which has been left over from previous reads. Initially b3 is assigned to leaf 9 as can be seen in the position map. When a request is made for this block, the path from the root to leaf 9 is read into the stash. The state of the system after this read can be seen in figure 3.1b, where the path that is read is highlighted in green. The figure also shows that block 9 is re-mapped to a randomly selected leaf. The path is then written back to the tree in a bottom-up fashion starting from the leaf. If there are any blocks in the stash that are assigned to the leaf bucket, they are evicted and the level just above is considered. This level can host any blocks that are assigned to any of the leafs among its children. In this case, bucket 2

Algorithm 1 Path ORAM pseudo-code

```
1:  procedure PathAccess(id, data, write)
2:      oldPos ← positionMapLookup(id)
3:      randomPos ← chooseRandomLeaf()
4:      positionMap(id) ← randomPos
5:      if oldPos == null then
6:          stash ← readPath(root, randomPos)
7:          oldPos ← randomPos
8:      else
9:          stash ← readPath(root, oldPos)
10:
11:     if write then
12:         update block in stash
13:     else
14:         return block from stash
15:
16:     WritePath(root, oldPos)
```

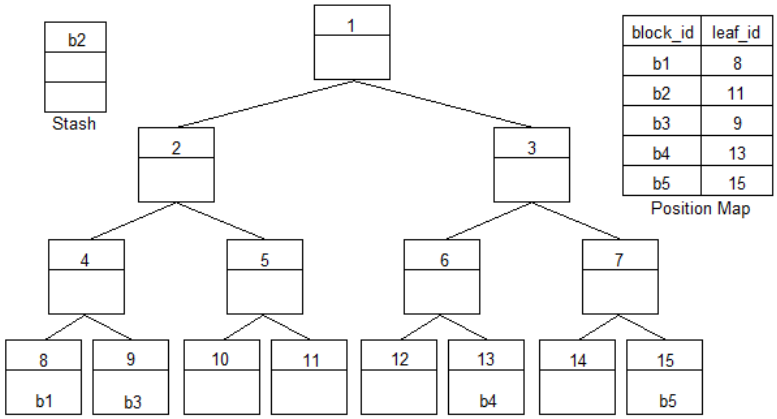
is eligible to receive blocks assigned to either of leafs 8, 9, 10, or 11 from the stash. This results in b2 being evicted into the bucket at that level. The root can actually receive blocks assigned to any of the leafs, since it is a direct ancestor of all of them. The block b3 is evicted into the root leaving an empty stash. The state after this operation can be observed in figure 3.1c.

3.2 Our Construction

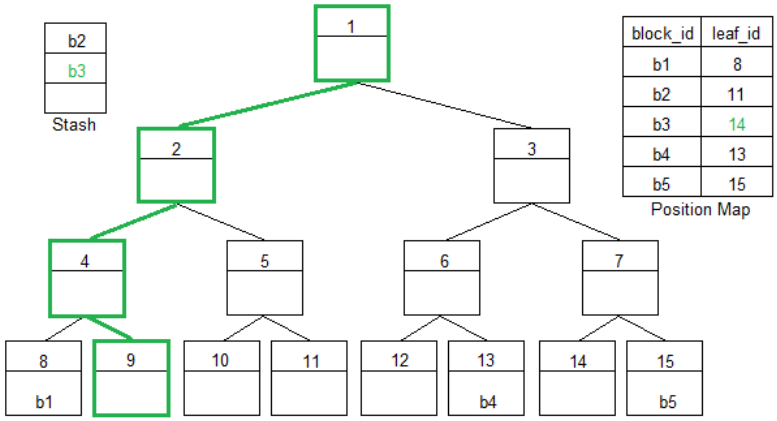
For a location-based service with a location indexed database storing user check-ins, the block identifier can be replaced with a location identifier. The payload of the block can contain the information associated with that check-in based on the functionality that the application aims to provide. The algorithm described above only works for the case where block ids are distinct and does not support duplicates. For most location-based services the database should be able to support multiple entries into the database for the same location. Modifying Path ORAM to support multiple check-in for the same location is possible without adversely impacting costs of the read path and write path operations.

Since Path ORAM is designed to read all the blocks that lie along a path into the stash,

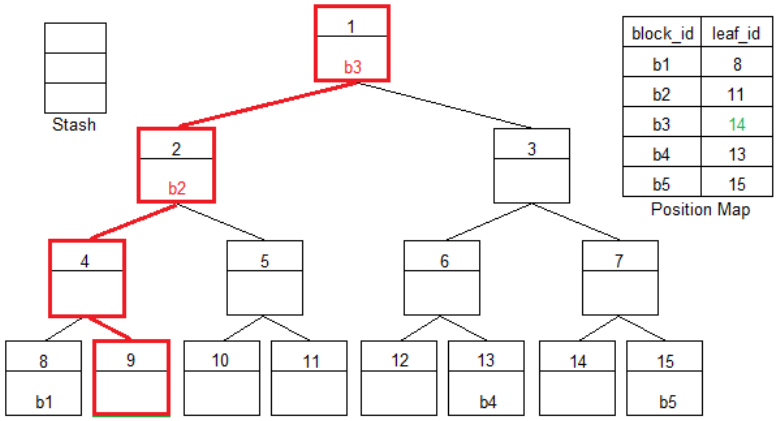
Figure 3.1: Path ORAM in action



(a) The initial state of the system.



(b) The state of the system when the path that block b3 lies on is read.



(c) The state of the system after the path is written back.

any check-ins that share the same location and hence the same identifier only have to be assigned to the same leaf bucket so that all of them lie on the same path. This ensures that all of these check-ins will be read into the stash every time a request is made. This does not require any modifications to the underlying ReadBucket() and WriteBucket(elements) operations. Furthermore, every location will have only one mapping in the position map no matter how many duplicates exist in the tree or the stash. As a result this does not have any adverse impact on the position map. As no changes are made to the way a path is read into the stash or written back, the obliviousness of this algorithm is preserved. Hence, the security analysis in [59] still applies. The ciphertext indistinguishability property of the encryption scheme ensures that the adversary is unaware of the existence of any duplicates.

One major change that we make is the use of a secure coprocessor to function as the client. The position map and the stash reside on the trusted memory in the secure coprocessor. As discussed in the Related Work Section, this turns the ORAM scheme into a PIR scheme. The secure coprocessor now manipulates the binary tree stored on the untrusted server to perform read and write operations. All communication that takes place with the secure coprocessor is encrypted with its public key to ensure secure communication over insecure channels of communication. Obviously, any requests made to the secure coprocessor have to be authenticated first.

Chapter 4

Design

We use an ORAM based design as that helps us fulfill our privacy goals described earlier. Since every update in an ORAM is indistinguishable from another, in a location-based service this means that an adversary is unable to distinguish between any requests coming in to the location databse. This also means that an adversary is unable to identify the location that a check-in is associated with. As a consequence even if an adversary observes the frequency of incoming check-ins and the time differene between them, tracking attacks are still futile. The rest of this chapter presents an overview of Muddler. Section 4.1 discusses the threat model. Section 4.2 describes the architecture of Muddler while Section 4.3 explains the public API exposed by the secure coprocessor.

4.1 Threat Model

This section discusses the threat model for this work. We assume that the entities in the system are honest-but-curious. Anyone may be interested in inferring additional information. The goal of Muddler is to prevent such an adversary from learning anything about users even if the adversary is able to obtain auxiliary information about users. No information should be leaked about any user's location. Our assumptions and aims are listed below:

- The system should not be vulnerable to any tracking attacks. The adversary should not be able to identify any of the locations that are part of any updates.

- The secure coprocessor is safe and secure. The secure coprocessor is completely trustworthy and can process user data without posing any threats, thus, also rendering it invulnerable to timing attacks. Furthermore, no adversary will attempt physically manhandling the device.
- The database is encrypted and only the secure coprocessor has access to the decryption key.
- Adversaries can observe the traffic between entities in the system.

The goal of our proposed solution is to let users check-in to any location, be it sensitive or public, without any qualms. However, we cannot account for cases where the venue owner is familiar with a user checking in. Our proposed solution does not consider any threats made possible by vulnerabilities in the apps on the user's smartphone or the device's operating system. We also do not provide any security guarantees against any harm caused by the user's negligence, such as physical loss of the device or theft. Any kinds of denial of service attacks are also beyond the scope of this research. We assume that standard operating procedures apply when it comes to such threats.

4.2 Architecture

This section presents an overview of the entities that Muddler is comprised of and how they relate to each other. Since our system is a location-based service, we have a database that is indexed by location. This database stores all information on user check-ins and does not provide any other functionality. This specific indexing scheme has been selected to provide optimal performance for the sort of queries that we provide through the API exposed by the secure coprocessor. This API is presented in Table 4.1. The service is composed of the following components:

- **The secure coprocessor:** The execution takes place on a secure coprocessor, which has all the characteristics of a trusted black box. Communication with the secure coprocessor takes place over a secure channel such as TLS. The secure coprocessor exposes an API, which is used to interact with the system. Only the secure coprocessor can access the location database and manipulate it while ensuring obliviousness.
- **The location-indexed database:** This is basically the ORAM storage. It is organized as a single data structure where every entry contains information about a

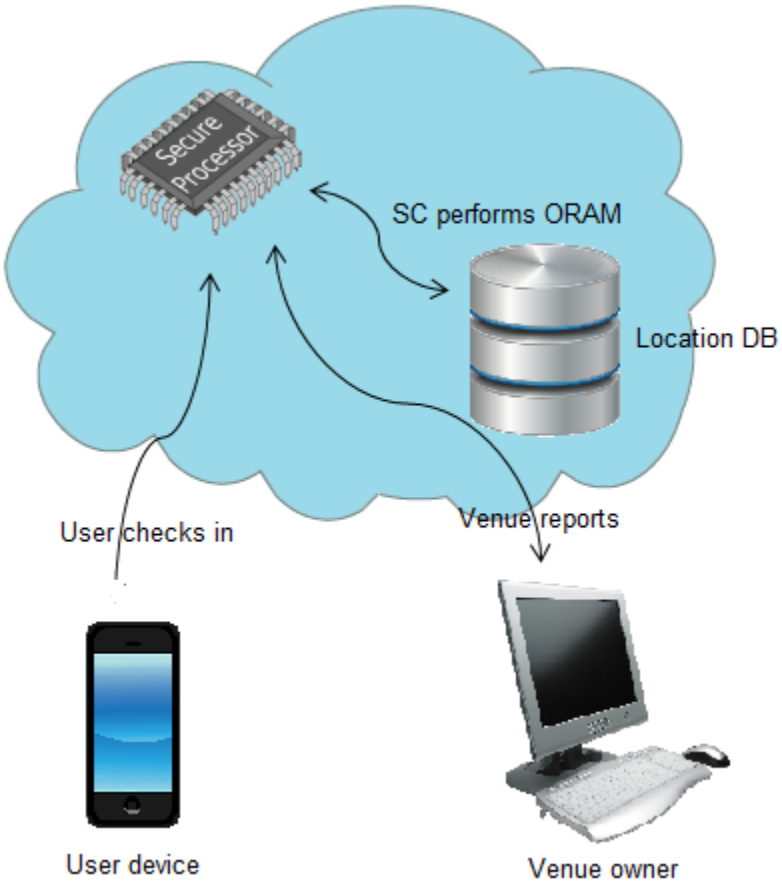


Figure 4.1: System overview

API provided by **SC**
`pathAccess(id, data, write)`
`totalCheckins` ← `findTotalCheckins(id)`
`uniqueVisitors` ← `findUniqueVisitors(id)`
`histogram` ← `findVisitorFrequency(id)`

Table 4.1: API provided by **SC**.

unique check-in. This information consists of the location identifier, the user id identifying the user who created the check-in, the timestamp at the time the check-in was created and a tip if the user decided to leave one. The information stored in this database is encrypted by the secure coprocessor using a symmetric key.

- **The user device:** This device runs the smartphone application, which allows users to check-in at a location and possibly leave tips for other users. Venue owners may use a computer or a smartphone to login and view reports. Any requests made will eventually invoke one of the API functions exposed by the secure coprocessor. These requests will be encrypted with the secure coprocessor's public key.

4.3 Public API

This section presents a listing of the proposed public API functions exposed by the secure coprocessor. It further explains how each of these API calls is implemented and discusses possible use cases. It is important to keep in mind that this is not an exhaustive list of the API functions that this architecture can support. In fact, the list is quite long. This section simply presents some examples making the point of how simple it can be to deliver these kind of services with such a design. All of these requests can only be made by authenticated users. The secure coprocessor supports the following:

- `pathAccess(id, data, write)`: The secure coprocessor performs an ORAM access operation as described in previous chapters. The **id** is the identifier of the location in this case. **data** represents the information to be stored and consists of the userid, the current timestamp and a tip left by the user. **write** indicates if the information is be added to the ORAM or if it is just a lookup. This request is made by user devices, when they check-in to a location. Tips are limited to a size of 200 characters. If a user does not leave a tip, then a dummy string of the same length is added to

the entry. If the user does leave a tip but that is smaller than 200 characters it is padded to make up the difference. Since, these queries may exhibit a variable latency, before the system is made publicly available an extensive performance check should be performed. The secure processor should respond to every request after the worst case possible time to ensure that traffic patterns do not exhibit any trends. This stands for all exposed API functions.

- `totalCheckins` ← `findTotalCheckins(id)`: This request is made by venue owners and is part of the business services functionality offered to make the proposed solution commercially viable. The **id** is the location identifier of the venue. The secure coprocessor performs an ORAM access (not a write). The result consists of all the entries in the database and the stash for the corresponding location. The total number of these entries is returned to the venue owner. For example, for a restaurant owner, this request would return the total number of check-ins for that location.

- `uniqueVisitors` ← `findUniqueVisitors(id)`: This request is also made by venue owners and is the second such request that is also part of the business services offered. The **id** is again the location identifier of the venue. The secure coprocessor performs an ORAM access. The results contains all the relevant entries. The secure coprocessor then scans all of them counting the total number of unique user ids. This number is then returned to the venue owner. For example, if the same restaurant owner makes this request, it will return the total number of unique visitors to that location.

- `histogram` ← `findVisitorFrequency(id)`: This is yet another of the requests that supports the business services offered by our location based service. When a business user makes this request, the secure coprocessor performs an ORAM access and scans the results. This time, instead of just counting the unique user ids, the frequency of their occurrence is also tracked. The results are then returned to the venue owner. If the restaurant owner makes this request, this will let him see the frequency of returning visitors. It is important to note here that the system will not reveal unique user identifiers.

Chapter 5

Implementation

We implemented the modified Path ORAM algorithm to demonstrate the feasibility of our proposal. This chapter discusses the implementation of the proposed construction. The implementation was done in Java, which was selected because of its simplicity and its WORA (Write once, run anywhere) characteristic [7]. Our implementation does not use a dedicated secure coprocessor, which is instead simulated in code. The code is written so that the architecture can support moving the secure coprocessor's functionality to a real device easily. Section 5.1 describes the composition of an Entry in detail. Section 5.2 describes the implementation of buckets, while Section 5.3 discusses the data structure used to implement it. Finally Section 5.4 discusses the design choices made while implementing the stash and Section 5.5 describes the implementation of the complete algorithm.

5.1 Entry

Every check-in is stored as an instance of the **Entry** class. This class is designed to represent a check-in instance. Every Entry consists of:

- **location**: This is an identifier of the location where this check-in took place. Initially we considered using GPS co-ordinates and complete location addresses, later changing this to an Integer representation to minimize any memory requirements imposed on the secure coprocessor. This is comparable to the venueId parameter that Foursquare includes in a check-in.
- **User ID**: This is a unique identifier assigned to every user of the service, stored as an Integer.

- **Tip**: This is the tip that a user may leave for others when they check in at a location. This is stored as a String and has a maximum length of 200 characters.
- **Time Stamp**: This is a time stamp that is stored with every check-in. This is used when dropping expired check-ins.

5.2 Bucket ORAM

Every instance of a bucket on the server is an instance of the BucketOram class. This contains an encrypted array of instances of the Entry class described above. Since every Entry written in the bucket is read into the stash, we encrypt the entire array and not individual entries. An empty bucket contains an array initialized with dummy entries. This array is then encrypted. The encryption used is AES with a key length of 128 bits in counter mode. As we encrypt the entire array of entries, we need to make sure that all entries have the same size so that unique buckets are indistinguishable based on their size. We make sure of that by padding entries with dummy bits so that they are of a constant size. Every BucketOram instance supports the following operations:

- **readEncrypted()**: This operation decrypts the stored array of entries and returns it.
- **writeEncrypted(*bucket*)**: This operation replaces the stored array of entries with *bucket* and then encrypts it.

5.3 Position Map

The position map is implemented using the Java TreeMap<Key, Value> [3] data structure. This is a Key, Value data store, implementing the Java Map interface, based on a Red-Black tree. This tree is sorted by keys and has a lookup cost of $O(\log N)$. The Key in this case is a representation of the location stored against the identifier of the leaf bucket that the location is assigned to. Since a location is represented as an Integer, this data structure is ideal as this enables several business intelligence operations through ranged queries. For example, if the service provider wants to look at the statistics of user check-ins for multiple locations over a block, this data structure is ideally suited for that.

5.4 Stash

The stash as described in Path ORAM [59] is a data structure that should be able to support range queries. A binary search tree is the standard data structure that may be used for that purpose. We however, take a different approach. When we started implementing our construction, we strictly followed Path ORAM. This meant that the construction was unable to support multiple check-ins for the same location. The stash was designed as a HashMap [2] of ArrayLists [1], HashMap < Integer, ArrayList < Entry > >. A HashMap is a hashtable based implementation of Java's Map interface, with an average case lookup cost of $O(1)$, while an ArrayList is a data structure that has an underlying array, which grows dynamically as the number of elements increases. The Key in this case being the identifier of the leaf bucket stored against a list of the check-ins at locations assigned to that leaf. During the write back operation the program only had to use the identifier of the leaf bucket to retrieve the ArrayList of all the check-ins at locations assigned to that leaf in the stash. The list would then be emptied as these check-ins were used to fill the buckets that lied along the path from the root to the leaf.

This implementation of the stash functioned well for preliminary tests, which were designed to test the validity of our assumptions. However, there was a problem. This approach was costly in realistic test scenarios, especially when we looked at multiple check-ins for the same location. Using a list to store check-ins that belonged to different locations all assigned to the same leaf necessitated the use of iteration over the entire list when writing back the path, searching for entries or when re-assigning entries to new locations. For locations that were very popular, this meant that the size of this list could become very large, impacting the latency of search or write-back operations.

To reduce this and to optimize the complexity of the stash, we looked into two design possibilities:

- HashMap<Integer, ArrayList< ArrayList < Entry> >: The key is still the identifier of the leaf but the value is an ArrayList of ArrayLists. This basically means that all check-ins at the same location are grouped into one ArrayList < Entry >. When a lookup takes place, the returned ArrayList contains multiple ArrayLists, each of them containing all the check-ins in the stash corresponding to one of the locations assigned to that leaf. This would still require iterating through this list during search operations to reach the list containing the check-ins of the location that is being searched for, although re-assignment becomes much simpler as the entire ArrayList representing one location can simply be removed and added to the new leaf node's

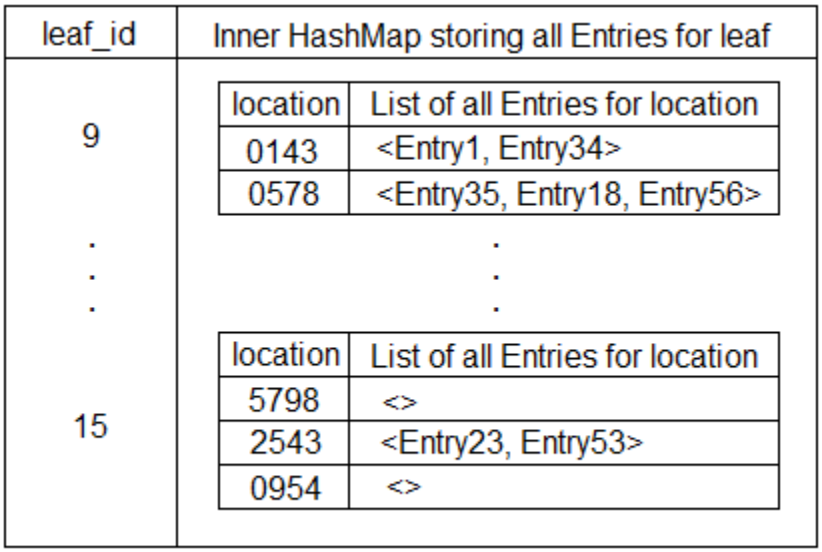


Figure 5.1: A pictorial view of the stash

List of locations during re-assignment thereby reducing the cost of this operation several times.

- HashMap<Integer, HashMap< Integer, ArrayList < Entry > > >: In this case the key is the identifier of the leaf but the value is a HashMap of ArrayLists. This internal HashMap maps location identifiers to the ArrayList < Entry > , which contains all the check-ins for that one location in the stash. For a search operation, this would only require two lookups of average cost $O(1)$ to reach the list of check-ins being searched for. The first lookup would return the internal HashMap containing all the check-ins of all the locations assigned to a certain leaf node. The second lookup would return the ArrayList containing all the check-ins for a certain location. During re-assignment this ArrayList can be removed from the old internal HashMap and added to the new leaf bucket's internal HashMap.

Whichever design we choose, we need to make sure it allows for the way the write back works in Path ORAM. It is quite clear that the second option is the most efficient option. In fact, this solution is even better than using a binary search tree as the underlying data structure for the stash. Figure 5.1 shows a pictorial view of the stash.

An interesting question is why we chose this design which essentially used two keys,

one for the first lookup and the other for the second lookup in the internal HashMap. We could have used one HashMap that combined both keys. If we combined both keys, the write back operation would not be possible. Currently all we need to do during write back is lookup the leaf identifier in the stash to obtain the internal HashMap. During write back we simply iterate over all the ArrayLists in this data structure, emptying them as we fill up the buckets being written back to the tree. If the combined keys were used we would need to store extra information somewhere since the keys would consist of both the leaf identifier and the location identifier.

5.5 Muddler

Now that we have described all the building blocks of our construction, this section explains how the PathAccess procedure now differs from Path ORAM. Algorithm 2 presents the pseudo-code of our construction. During initialization a value is assigned to an expirationTimer. For example, if we are interested in only storing check-ins for a maximum of one month then that is the value that will be assigned to this timer. When PathAccess is now invoked, it looks up $Entry1$'s location in the positionMap to find out the leaf that it is assigned to. It then generates a $randomPos$ signifying one randomly chosen leaf. If $oldPos$, or the leaf value is null, this means that the location does not exist in the system. In this case, the path from the root to the randomly chosen leaf is read into the stash. If this is a write access, then a stash look up is performed against $randomPos$. This returns the $randomLeafMap$. This data structure contains all the check-ins in the stash for the randomly selected leaf. As the location does not exist, a new ArrayList $<$ Entry $>$ is created and $Entry1$ is added to this List. This List is then inserted into the $randomLeafMap$, against $Entry1$'s location identifier. The $positionMap$ is updated to reflect the position of $Entry1$ and the path is written back.

If $oldPos$ is not null, this means that the location already exists in the system. In this case the path from the root to the $oldPos$ is read into the stash. A stash lookup is performed to retrieve $oldLeafMap$ containing all the check-ins in the stash for all locations assigned to the leaf $oldPos$. Another stash lookup is performed to retrieve $randomLeafMap$, which contains all the check-ins in the stash for the randomly selected leaf. $oldList$ corresponds to the ArrayList $<$ Entry $>$ containing all check-ins for $Entry1$'s location in the stash. This list is removed from $oldLeafMap$. If this is a write operation then, Entry1 is added to this list. This list is then inserted into the randomLeafMap against $Entry1$'s location identifier. The $positionMap$ is updated to reflect the position of $Entry1$ and the path is written back.

It is important to note that while a path is being read into the stash, the timestamp of every Entry is compared with the expirationTimer to make sure the Entry has not yet expired. Similarly, when a path is being written back to the server, the timestamp of every Entry is compared with the expirationTimer. All expired check-ins are dropped and not added to the stash. A periodic eviction takes place that scans the entire stash and removes any expired check-ins that still remain in the stash. For the purpose of our experiments in Section 6, we invoke this subroutine once a day. The frequency of this invocation should vary depending upon the context in which Muddler is used.

Algorithm 2 Muddler pseudo-code

```
    procedure PathAccess(Entry1, write)
 2:     oldPos ← positionMapLookup(Entry1.location)
        randomPos ← chooseRandomLeaf()
 4:     if oldPos == null then
            stash ← readPath(root, randomPos)
 6:         if write then
                randomLeafMap ← stashLookup(randomPos)
 8:             create new ArrayList<Entry> and add Entry1
                insert this list into randomLeafMap against Entry1.location
10:         oldPos ← randomPos
        else
12:         stash ← readPath(root, oldPos)
            oldLeafMap ← stashLookUp(oldPos)
14:         randomLeafMap ← stashLookUp(randomPos)
            oldList ← oldLeafMap.remove(Entry1.location)
16:         if write then
                add Entry1 to oldList
18:         Insert oldList into randomLeafMap against Entry1.location
        positionMap(Entry1.location) ← randomPos
20:
        WritePath(root, oldPos)
```

Chapter 6

Experiments

This sections presents the experiments we conducted and describes our results. Section 6.1 describes the experimental setup that we used. Section 6.2 discusses the results of our performance analysis, while Section 6.3 explains in detail how we created our realistic check-in data set and the subsequent set of experiments and results. Our results show that our proposed design is a feasible solution that improves on the state of the art in privacy preserving frameworks for location-based services.

6.1 Setup

The experiments were conducted on the CrySP [4] RIPPLE Facility [16], which is a set of very powerful machines dedicated to research in Privacy Enhancing Technologies. The machine used for the purpose consists of 80 physical cores. With Hyper-Threading they amount to 160. The machines also boasts 1 TB of Random Access Memory. The operating system used was Ubuntu Linux.

6.2 Performance Analysis

6.2.1 Empirical Stash Size Analysis

The first set of experiments we conducted were intent on determining the probability of the occurrence of varying stash sizes. Thus, an empirical analysis was performed. The data set

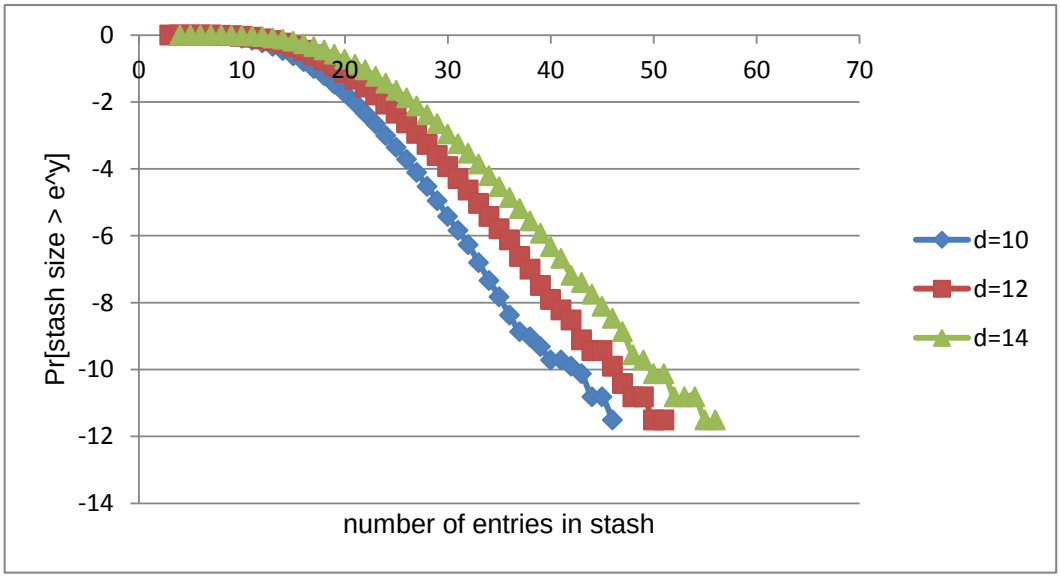


Figure 6.1: Stash size distribution for worst-case ORAM accesses.

used for this purpose represented the worst-case access traces [59], exhibiting no locality and a lack of any entries with duplicate block identifiers. Entries are accessed sequentially, wrapping around the last one. We conducted these experiments for trees of varying depths. Tree capacity was fixed at 50% of the maximum number of entries the tree could possibly support [42]. The experiments were structured as follows:

- Start with an empty ORAM and perform N accesses to place all N entries into the tree.
- Perform 5 million worst-case ORAM accesses to warm up the ORAM.
- Perform 100,000 ORAM accesses and record the stash size every time a path is read into the stash.

For each tree configuration used, Figure 6.1 plots the probability of the stash size being greater than a certain value. This includes the number of entries read into the stash during reads. As can be seen, the trends compare fairly well with the results in [42]. It is quite clear that the probability of the larger stash sizes recorded is actually quite low.

Depth	Bucket = 4	Bucket = 8	Bucket = 16	Bucket = 32	Bucket = 48
10	4,096	8,192	16,384	32,768	49,152
12	16,384	32,768	65,536	131,072	196,608
14	65,536	131,072	262,144	524,288	786,432

Table 6.1: Tree occupancy for varying configurations

6.2.2 Latency

The second set of experiments that we focused on includes recording the average latency of an access and an empirical analysis of how this latency is distributed over the various core operations in Muddler. The tree occupancy was again maintained at a level half that of its maximum capacity. Varying tree configurations were used to figure out the impact of increasing tree capacity through either increasing bucket size or tree depth or both since this is a very obvious design concern. As we are using binary trees, an increase in the depth of the tree by one level means that the capacity of the tree has doubled. Similarly, if we double the bucket size the impact on the capacity of the tree will be similar without any impact on the depth of the tree. Table 6.1 shows the number of entries that the various tree configurations support. This set of experiments also used the worst-case access traces and was structured as follows:

- Start with an empty ORAM and perform N accesses to place all N entries into the tree.
- Perform at least 100,000 worst-case ORAM accesses to warm up the ORAM.
- Perform 100,000 ORAM accesses and record the operational statistics for each of them.

Figure 6.2 is a plot of latency against bucket sizes. The figure shows how the latency of an access increases as the bucket size increases, when the depth stays invariant. The plot depicts the trend for various bucket sizes. It is quite clear that the impact of increasing bucket size to increase the capacity of the tree exhibits a linear increase in the latency of the access operation. This makes sense intuitively since the increase in the number of operations including encryption, decryption and stash eviction should be somewhat proportionate to the increase in capacity. Figure 6.3, which is a plot of the total encryption and decryption cost for each of these bucket sizes, and Figure 6.4, which plots the total

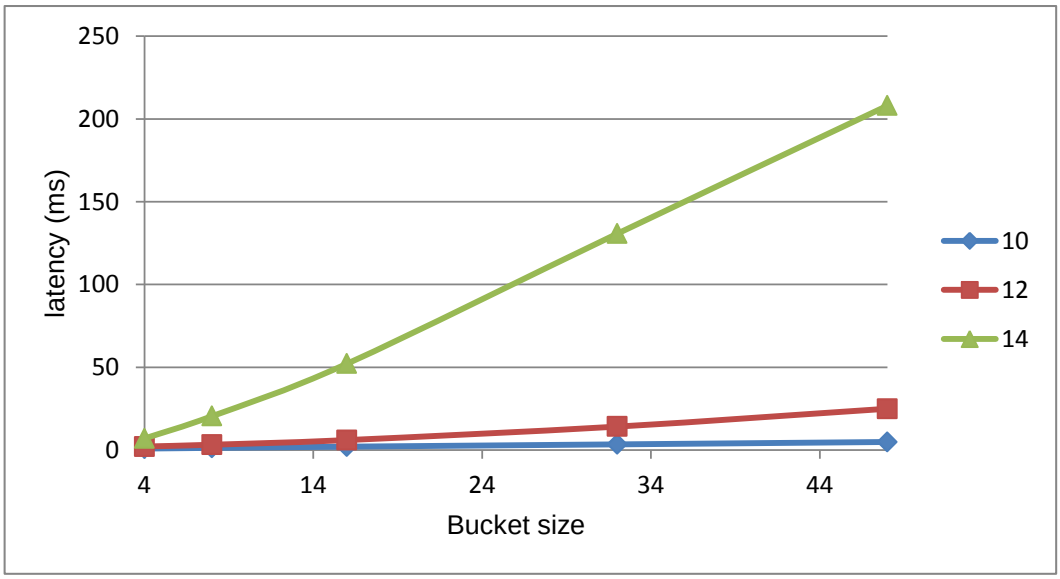


Figure 6.2: Plot of latency against bucket size where depth does not vary.

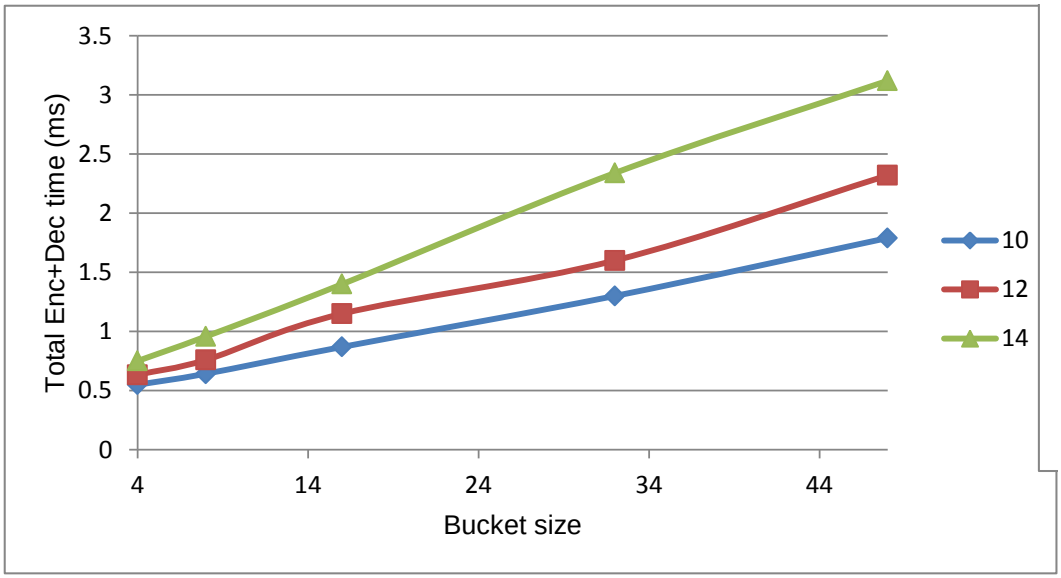


Figure 6.3: Plot of total encryption and decryption cost against bucket size where depth does not vary.

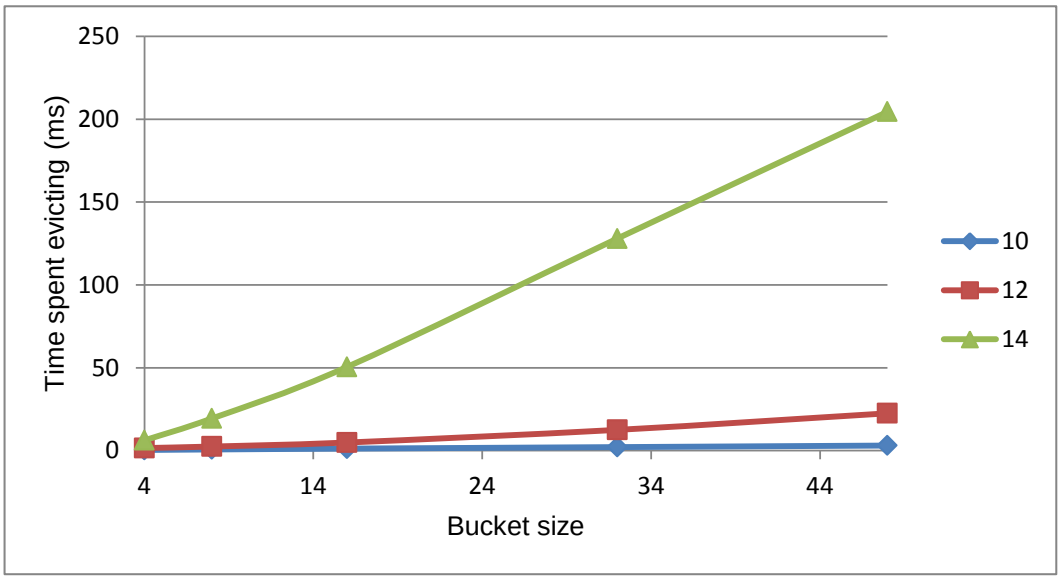


Figure 6.4: Plot of time spent evicting entries from the stash while writing a path back against bucket size where depth does not vary.

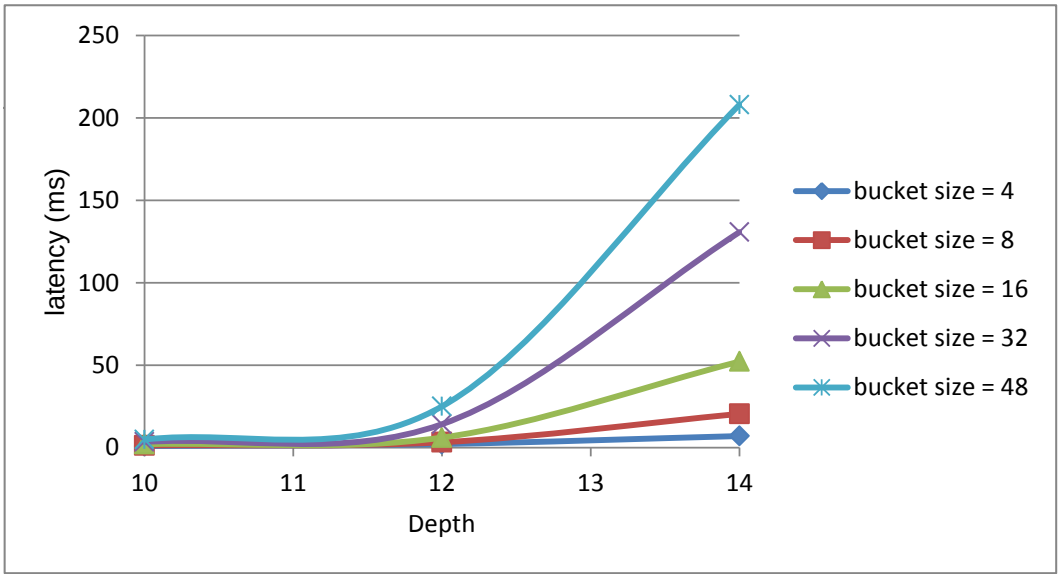


Figure 6.5: Plot of latency against depth where bucket sizes do not vary.

time spent in the write path operation scanning the stash and evicting entries into the tree substantiate this assertion.

On the other hand, Figure 6.5, plots latency against varying depths. Bucket sizes remain unchanged as depths increase. It can be clearly observed that the impact on latency is far from linear. As a result we realize that even if initially increasing the depth of the tree seems to present an option with less latency increase, at some point in time the increase will far outstrip the cost imposed by increasing the bucket size. This does raise a few questions. What contributes to this increase? When does it become more feasible to increase bucket size against depth? To answer this question we look at the contribution towards latency of the two most likely culprits. This is a reference to the total of the encryption and decryption overhead and the overhead imposed by the stash eviction procedure that is built into every single access. We record the total time spent in each of these operations and use them for our analysis.

Figure 6.6 is a plot of the total cost of encryption and decryption averaged over 100,000 accesses as a percentage of total cost. It is quite clear that as the depth of the tree increases, the contribution of encryption and decryption towards the total latency decreases substantially. Interestingly enough, Figure 6.7 shows that almost all of this difference is picked up by the core procedure writing back the path. If we look at both the two graphs, we see that, even for a relatively small tree with depth 10 the total contribution of encryption and decryption and write back easily amounts to more than 90% of total cost. As the depth of the tree increases, the lion's share of the cost is contributed by the time spent evicting entries from the stash into the tree during the write path method. This should not be surprising. The way Path ORAM is designed, as you get closer to the root, the number of leaves that are the children of the current node increases. Not all of these nodes have entries waiting to be evicted in the stash. This usually ends up forcing Path ORAM to scan most of these children. The root can host entries from any of the leaf nodes. During our empirical analysis we noticed that, for the tree sizes we were working with, most leaf nodes were accessed for every single access. This is apparently the greatest bottleneck visible in our results.

As for when is it more feasible to choose an increase of depth over an increase of bucket size, the answer is not so simple. It depends on the constraints of the system and the design requirements. For example, a tree of depth 17 and bucket size 4 can accommodate a little over half a million entries at 50% occupancy and the latency is 202 milliseconds. A tree of depth 14 with bucket size 32 on the other hand can accommodate a similar number of entries at 50% occupancy with a latency of 130.7 milliseconds. Latency is not the only difference between the two trees. Another difference is the fact that the maximum number of entries that can lie along a path is not even comparable. The first tree can support only

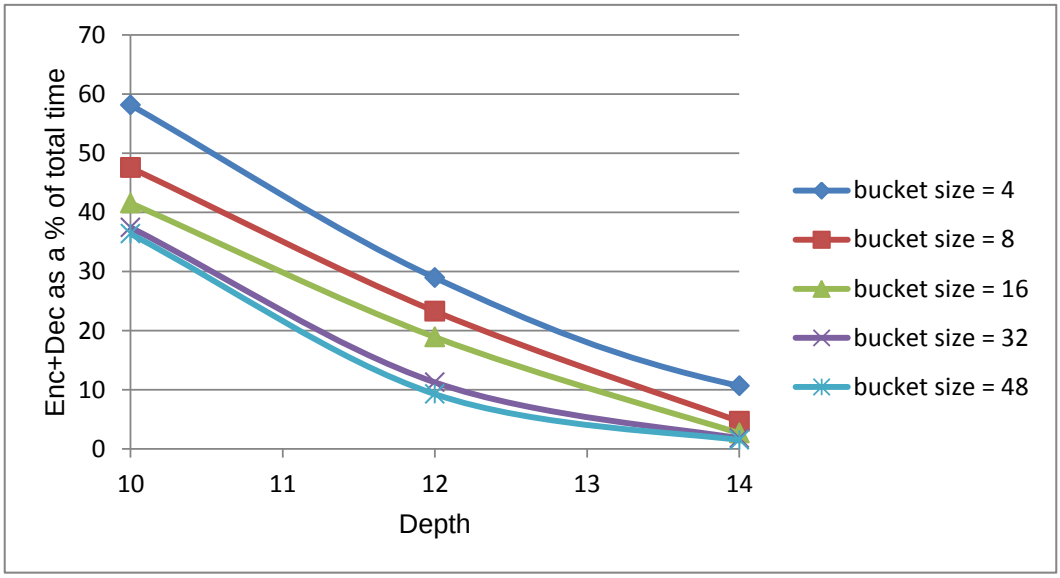


Figure 6.6: Plot of total encryption and decryption cost as a percentage of total cost for varying depths when bucket sizes do not vary.

a maximum number of $17 * 4 = 68$ entries along a path, while the second can support a maximum of $14 * 32 = 448$ entries along a path. For the scenario we are interested in it seems we should be more interested in paths with higher maximum capacity as compared to a greater number of paths with less capacity.

6.3 Real World Data

Ideally we would have wanted to perform our experiments using real world check-in data from a size-able service. Not surprisingly, such a data-set is not readily available. Contacting Foursquare did not help with the service interested in avoiding collaboration with researchers in privacy. Most of the research requiring similar data-sets for experimentation has used a subset of real-world data. The most common approaches towards data gathering can be summarized as follows:

- Scraping [41, 53]: In both cases data was collected over a time period of several months. Using this approach to collect venue data has several drawbacks. First of all, Foursquare rate limits queries to its servers. An application is allowed to make up

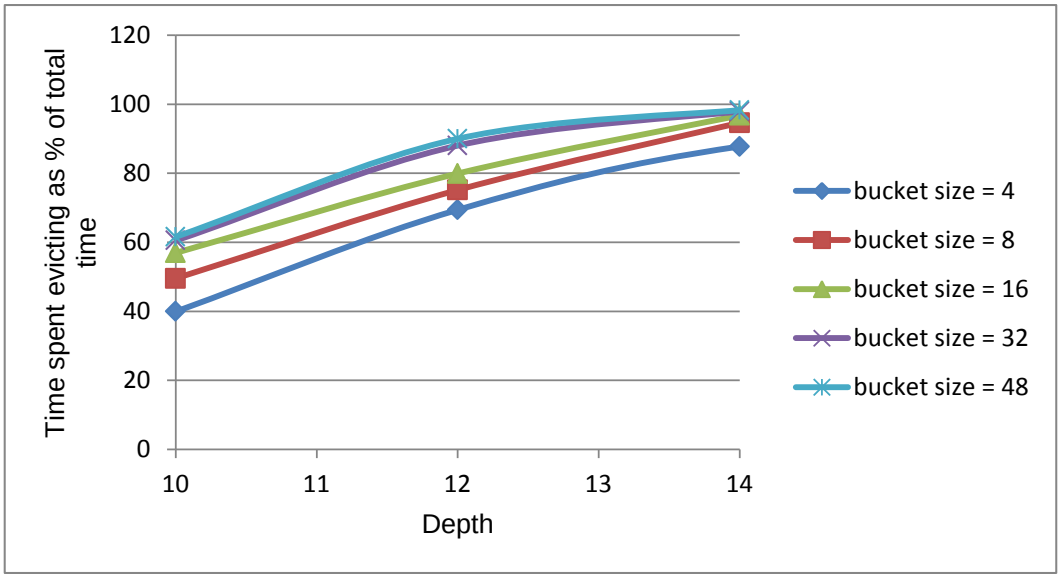


Figure 6.7: Plot of total time spent evicting entries from the stash while writing a path back against depth when bucket sizes do not vary.

to 500 authenticated requests per hour. Furthermore, the API is also not designed to support scraping queries that try to access information on all the venues in a region. As a result, this approach requires using multiple machines collecting data over a long time period, making requests over very small regions to make sure all venues are covered. Even so, there is no guarantee that the entire set will be covered or will cover accurate data since Foursquare always "fuzzes" home venue location information for third party applications irrespective of who makes the request.

- Collecting public check-in data from Twitter: This approach has been used by several works such as [20, 56, 19, 46] and has an inherent flaw. It can only collect information that has been shared on Twitter. This is necessarily a subset of the total set of check-ins. Furthermore, the data might be spread over a huge geographical area making it difficult to model the check-in distribution of a small contiguous region such as a town.

- Using cellular data [45]: Every time a cellular service subscriber makes an interaction with the telecommunication service, the location of the user can be approximated within the vicinity of a service tower. Since, it takes two to perform an interaction over the telecommunication service, every interaction provides information on two

Airport	Avg. daily check-ins	Boardings 2013
JFK International Airport	1164	25,036,855
LaGuardia Airport	792.2	13,353,365
Lambert - St. Louis International Airport	216	6,213,972

Table 6.2: Average number of daily check-ins at international airports

users. This data has also been used by researchers but would definitely necessitate collaborating with a telecommunication service due to privacy concerns.

This is just a summary of the more popular means of gaining access to large scale user location data. One of the more quirky approaches have been the proposal of using mobility traces from virtual worlds such as Second Life [38]. However, none of these approaches suit our purpose.

One interesting observation by Li et al. [41] is that venues that can be categorized in the Travel and Transportation category are among the most popular and record among the highest number of check-ins per venue. Another important observation in the same work is that the distribution of public check-ins at individual venues is Zipfian in nature. Zipf's law is a power law, which in its simplest form in this context states that the most popular venue will have twice as many check-ins as the second most popular one. Simply put, if n is the number of check-ins at the most popular location, the number of check-ins at the second most popular location will be $n/2$, the number of check-ins at the third most popular location will be $n/3$ and so on.

Unable to obtain access to a real world data set with complete check-in information over geographical regions of any size, we use these observations to our advantage. We formulated a list of all the major hubs of transportation in two cities namely, New York City and St. Louis. Our list consists of all airports, bus terminals and train stations. We wrote a crawler in Java using jsoup, a Java based HTML Parser. Starting the 26th of April up to the 2nd of August 2014 we collected the total number of check-ins from the Foursquare page of all these venues at hourly intervals. Based on our observations, we can confidently state that airports tend to be among the most popular locations in cities. The average daily number of check-ins at the three US airports in the cities of New York and St. Louis can be seen in Table 6.2.

Now that we have the numbers for these three airports and we know that airports tend to be the most popular venues for check-ins we can model distributions for the cities of New York and St. Louis based on the simple Zipfian distribution. The question still remains how do we use this information to model much larger check-in distributions?

City	Boardings 2013	Predicted avg. daily check-ins
New York City	25,036,855	1,217
Buffalo	2,568,018	126
Rochester	1,209,532	60
Albany	1,196,753	60
Syracuse	991,663	50
White Plains	770,550	39
Islip	662,612	34
Newburgh	163,815	10
Plattsburgh	151,235	9
Elmira	129,749	8
Ithaca	103,722	7
Niagara Falls	98,958	7
Binghampton	95,210	6
Watertown	18,818	3

Table 6.3: Predicted number of average daily check-ins at airports in New York state.

Since, the number of check-ins at an airport is dependent on the number of visitors to these venues we come up with a very naive approach to find a relation between the number of check-ins at different airports. The Federal Aviation Administration maintains passenger boarding information at airports within the US. A ranking of all commercial service airports ranked by number of passenger boarding in the year 2013 is available at [5]. A commercial service airport is defined as any public airport with more than 2499 passengers boarding in a year.

Now that we have the number of check-ins at some airports and their passenger boarding numbers, a very naive approach to simulate the numbers for other cities is to use these few data points to come up with a regression, which can then be used to predict the number of check-ins at other airports. The data points to be used for this regression are tabulated in 6.2. Using the average number of daily check-ins to represent the y coordinates and the number of boardings at that airport in 2013 to represent the x coordinates the linear regression that we obtain is:

$$y = 4.854726463 \times 10^{-5}x + 2.262829062$$

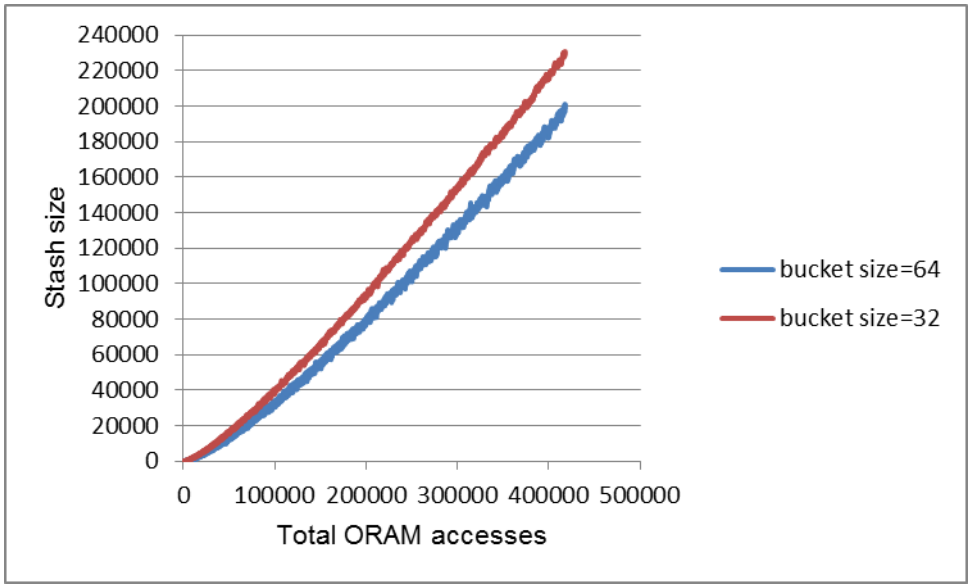


Figure 6.8: Plot of stash sizes when no expiration of check-ins takes place.

The predicted average daily check-in counts for all commercial service airports in the state of New York are tabulated in 6.3. The table also shows the number of boardings that took place at these airports within the year 2013, which are used for this prediction. LaGuardia is not included in this list, since only one airport from every city is considered. From the city of New York, the only airport we consider is the John F. Kennedy International Airport.

Now that we have a somewhat realistic estimate of daily check-ins at these New York airports, we can use the Zipfian distribution to simulate the check-ins for these cities over a two week period. The total check-in count for the state of New York in this time frame comes to be somewhere around 209,000. If we are to conduct an experiment for a period of four weeks that would create almost double the number of check-ins. If we look at Table 6.1 again we can see that a tree of depth 14 with a bucket size of 32 suits our requirements perfectly. So for our first experiment we create such a tree and also create the check-ins for a four week period. We then shuffle these check-ins so that they seem to be coming in randomly and not sequentially. We then perform ORAM accesses to place these check-ins into the tree.

Figure 6.8 shows the results of this experiment. It is a plot of the stash size recorded when a path is read into the stash on the y-axis against the number of ORAM accesses that have been performed in total on the x-axis. During this set of experiments we did

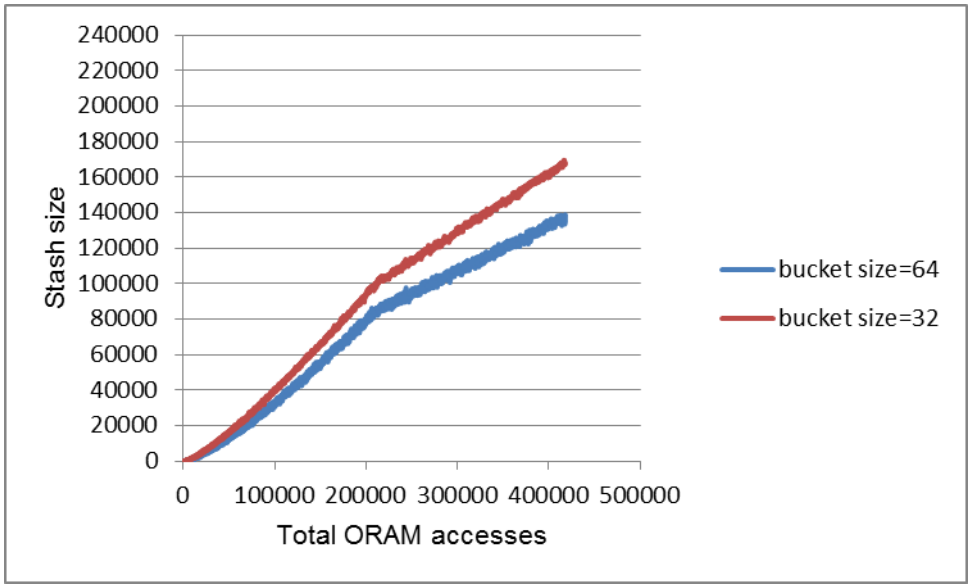


Figure 6.9: Plot of stash sizes when expired check-ins are dropped during reads and writes only.

not impose an expiration period over check-ins. As we can see the stash size for this tree keeps on increasing into the hundreds of thousands. Out of curiosity we conducted the same experiments with a tree with a similar configuration except it had a bucket size of 64. As expected, the maximum stash sizes for this tree are noticeably less than the previous observation. This tree does impose some obvious disk storage requirements, which would double simply because it has double the capacity of the tree with bucket size 32.

The very large stash size requirement exhibited in our first set of experiments with realistic data can be justified by the presence of very few very popular locations and a lot of other locations which receive much fewer check-ins. Not all the check-ins for these popular locations can be accommodated in the tree, since one location can only be assigned to one path. If the number of check-ins for that location exceeds the capacity of that path, those check-ins will then stay in the stash. For example, a tree with depth 14 and bucket size 32 can only accommodate a maximum of 448 entries in on path from the root the leaf, while some of the most popular location have way more check-ins than that as shown in Table 6.3. These check-ins end up staying in the stash.

Now, we test the performance of our expiration scheme with a rolling expiration window. A two week expiration window means that all check-ins older than two weeks at that point in time have expired. While performing a read the timestamps of all the check-

ins are compared with the current time. If the check-in has expired it is not read into the stash. Similarly, during a write back if an expired check-in is encountered it is not written to the ORAM and is dropped. Figure 6.9 shows the results of this second set of experiments. Interestingly, both Figures 6.8 and 6.9 reveal that the rate at which the stash size increases for the tree with smaller capacity is much higher. This is not surprising given the distribution of check-ins and the tree capacities.

It seems that once the ORAM enters into the time period where expiration starts, the line reflecting the fluctuation of stash sizes abruptly becomes much less steeper. This basically reflects a slowdown in the rate at which the stash size increases. Ideally, this should have exhibited some sort of a stabilizing trend in the stash size, since older check-ins should be expiring at a rate somewhat similar to the arrival rate of new check-ins but that is clearly not happening. The reason for this is the fact that not all expired check-ins are being dropped. During reads and writes, some expired check-ins are definitely dropped but it seems that quite a large number of the check-ins for most likely the most popular of locations are never moved between the stash and the tree, which is why they are never dropped during reads and writes. This necessarily imposes some unjustified constraints on the secure coprocessor since that is where the stash is contained. The larger the stash, the higher the memory requirements for the secure coprocessor.

A very obvious solution to this problem is the addition of a background eviction subroutine. This subroutine once invoked traverses the entire stash accessing every element. If an expired check-in is encountered that is removed. This subroutine necessarily imposes its own latency requirements but as the rate of daily check-ins is only in the order of tens of thousands and the stash size after the initial two week period hovers around the 80,000-100,000 mark, it seems that invoking this subroutine once a day should be reasonable. Besides, dropping check-ins while reading and writing has already proven to inhibit the growth rate of the stash size. Figure 6.10 shows a plot of the stash sizes when expired check-ins are dropped by once a day invocation of the eviction subroutine and dropping expired check-ins during reads and writes. The results show that the stash size does in fact stop increasing and stabilizes. The results also show that the number of elements in the stash at all points in time stays well below half of all the real entries. The rest can be found in the tree interspersed with dummies. As secure coprocessors have limited resources our results demonstrate the stash size requirements for an instantiation of Muddler serving the state of New York.

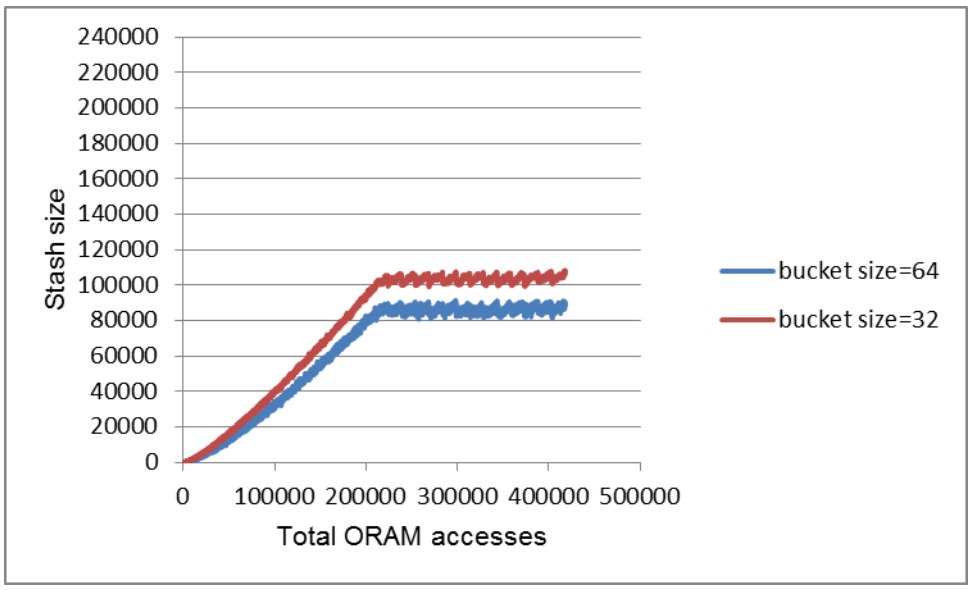


Figure 6.10: Plot of stash sizes when expired check-ins are dropped during reads and writes and eviction takes place periodically.

Chapter 7

Future Work

This chapter discusses the possible future work. We foresee future work primarily along the following themes:

- **Real world check-in data:** Our experiments were performed using predicted check-in data. We understand that the distributions that we use in our current set of experiments may not very accurately depict reality. It would be interesting to see how the stash sizes fare when a real world check-in data set is used, provided it is complete.
- **API extension:** The current API only exposes a few functions since our goal was primarily to provide a proof of concept implementation. This API can be very easily extended. For example, features such as a tip search for specific locations is one of the possible new features that can be added, enabling visitors to search all the tips that have been left for a certain location by previous visitors. Another such extension can be the introduction of mayorships where users that visit a location the most become the mayor of that place, yet another feature that the new Foursquare swarm application offers.
- **Secure coprocessor:** Currently our implementation does not use a physical secure coprocessor, although it should not be difficult to move modules of our code to a real device. Possible future work can also include the use of a real secure coprocessor and an analysis of how the limited power and memory of that device impacts performance.
- **Applications:** Our current design is a very specific service but the design is quite flexible and can be easily extended to support a variety of use cases and services. For

example the architecture we propose can be very easily modified and deployed as a privacy preserving location-based behavioral advertising platform. Other such applications possibly include location-based alerts or even a geo-social network. Future work can involve a study into the design and performance trade-offs each of these applications would entail.

- **Design space exploration:** Our current implementation only uses a single tree data structure. Future work can also involve a design space exploration for various tree configurations even looking into the use of multiple trees being accessed concurrently. For example, our check-in data only represented the state of New York. If we want a system that supports a much larger geographic region we will either have to use an enormous tree, which would definitely have an adverse impact on latency or we could use multiple trees catering to smaller regions.

Chapter 8

Conclusions

Location-based services such as Foursquare are becoming increasingly popular. These services collect user location information, which raises serious privacy concerns. We extend Path ORAM, an Oblivious RAM protocol to provide support for multiple blocks with the same identifier. When used in conjunction with a secure coprocessor this creates a Private Information Retrieval scheme designed to ensure the location privacy of the users of Muddler, a privacy preserving location-based service. We also propose and discuss in detail the public API that is exposed and show that it is extensible and serves to improve the commercial viability of the design. We implement this design and present a performance analysis to prove its feasibility. Furthermore, we generate a realistic check-in data set for the state of New York and perform experiments. Our results show that the probability of failure of this design is based on the stash size requirements and the memory constraints imposed by the secure coprocessor. An interesting observation is that the greatest factor contributing to stash requirements is the distribution of the data set. We also show how various tree configurations pose different stash size requirements for the same data set. Our results show that Muddler is truly practical.

References

[1] Class Arraylist<E>. *http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html*, 2014.

[2] Class HashMap<K,V>. *http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html*, 2014.

[3] Class TreeMap<K,V>. *http://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html*, 2014.

[4] CrySP. *https://crysp.uwaterloo.ca/*, 2014.

[5] Passenger Boarding (Enplanement) and All-Cargo Data for U.S. Airports. *http://www.faa.gov/airports/planning_capacity/passenger_allcargo_stats/passenger/*, 2014.

[6] Raising awareness about over-sharing. *http://pleaserobme.com/*, 2014.

[7] Write once, run anywhere? *http://www.computerweekly.com/feature/Write-once-run-anywhere*, 2014.

[8] Michael Backes, Aniket Kate, Matteo Maffei, and Kim Pecina. Obliviad: Provably secure and practical online behavioral advertising. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 257–271. IEEE, 2012.

[9] Alastair R Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, pages 49–54. ACM, 2011.

[10] Alastair R Beresford and Frank Stajano. Location privacy in pervasive computing. *Pervasive Computing, IEEE*, 2(1):46–55, 2003.

[11] Bogdan Carbunar, Radu Sion, Rahul Potharaju, and Moussa Ehsan. The shy mayor: Private badges in geosocial networks. In *Applied Cryptography and Network Security*, pages 436–454. Springer, 2012.

[12] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.

[13] Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure ORAM with $O(log^2 n)$ overhead. *CoRR, abs/1307.3699*, 2013.

[14] Kai-Min Chung and Rafael Pass. A simple ORAM. Technical report, DTIC Document, 2013.

[15] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Communications of the ACM*, 57(3):99–106, 2014.

[16] CrySP RIPPLE Facility. https://ripple.uwaterloo.ca/, 2014.

[17] Kassem Fawaz and Kang G Shin. Location privacy protection for smartphone users. 2014.

[18] Foursquare. https://foursquare.com/, 2014.

[19] Huiji Gao, Jiliang Tang, and Huan Liu. Exploring social-historical ties on location-based social networks. In *ICWSM*, 2012.

[20] Huiji Gao, Jiliang Tang, and Huan Liu. gscorr: modeling geo-social correlations for new check-ins on location-based social networks. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 1582–1586. ACM, 2012.

[21] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 182–194. ACM, 1987.

[22] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[23] Michael T Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *Automata, Languages and Programming*, pages 576–587. Springer, 2011.

[24] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 95–100. ACM, 2011.

[25] Marco Gruteser and Dirk Grunwald. Anonymous usage of location-based services through spatial and temporal cloaking. In *Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 31–42. ACM, 2003.

[26] Saikat Guha, Mudit Jain, and Venkata N Padmanabhan. Koi: A location-privacy platform for smartphone apps. In *NSDI*, pages 183–196, 2012.

[27] Baik Hoh, Marco Gruteser, Hui Xiong, and Ansaf Alrabady. Enhancing security and privacy in traffic-monitoring systems. *Pervasive Computing, IEEE*, 5(4):38–46, 2006.

[28] Baik Hoh, Marco Gruteser, Hui Xiong, and Ansaf Alrabady. Preserving privacy in gps traces via uncertainty-aware path cloaking. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 161–171. ACM, 2007.

[29] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 639–652. ACM, 2011.

[30] IBM. IBM PCIe Cryptographic Coprocessor. *http://www-03.ibm.com/security/cryptocards/pciecc/overview.shtml*, 2014.

[31] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.

[32] Sharad Jaiswal and Animesh Nandi. Trust no one: a decentralized matching service for privacy in location based services. In *Proceedings of the second ACM SIGCOMM workshop on Networking, systems, and applications on mobile handhelds*, pages 51–56. ACM, 2010.

[33] Ali Khoshgozaran and Cyrus Shahabi. Private buddy search: Enabling private spatial queries in social networks. In *Computational Science and Engineering, 2009. CSE'09. International Conference on*, volume 4, pages 166–173. IEEE, 2009.

[34] Hidetoshi Kido, Yutaka Yanagisawa, and Tetsuji Satoh. An anonymous communication technique using dummies for location-based services. In *Pervasive Services, 2005. ICPS'05. Proceedings. International Conference on*, pages 88–97. IEEE, 2005.

[35] John Krumm. Inference attacks on location tracks. In *Pervasive Computing*, pages 127–143. Springer, 2007.

[36] John Krumm. A survey of computational location privacy. *Personal and Ubiquitous Computing*, 13(6):391–399, 2009.

[37] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in) security of hash-based oblivious RAM and a new balancing scheme. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 143–156. SIAM, 2012.

[38] Chi-Anh La and Pietro Michiardi. Characterizing user mobility in second life. In *Proceedings of the first workshop on Online social networks*, pages 79–84. ACM, 2008.

[39] Ulf Leonhardt and Jeff Magee. Security considerations for a distributed location service. *Journal of Network and Systems Management*, 6(1):51–70, 1998.

[40] Justin J Levandoski, Mohamed Sarwat, Ahmed Eldawy, and Mohamed F Mokbel. LARS: A location-aware recommender system. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 450–461. IEEE, 2012.

[41] Yanhua Li, Moritz Steiner, Limin Wang, Zhi-Li Zhang, and Jie Bao. Exploring venue popularity in foursquare. In *INFOCOM, 2013 Proceedings IEEE*, pages 3357–3362. IEEE, 2013.

[42] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 311–324. ACM, 2013.

[43] Google Maps. https://www.google.ca/maps/preview, 2014.

[44] Arvind Narayanan, Narendran Thiagarajan, Mugdha Lakhani, Michael Hamburg, and Dan Boneh. Location privacy via private proximity testing. In *NDSS*, 2011.

[45] Anastasios Noulas, Cecilia Mascolo, and Enrique Frias-Martinez. Exploiting foursquare and cellular data to infer user activity in urban environments. In *Mobile Data Management (MDM), 2013 IEEE 14th International Conference on*, volume 1, pages 167–176. IEEE, 2013.

[46] Anastasios Noulas, Salvatore Scellato, Neal Lathia, and Cecilia Mascolo. A random walk around the city: New venue recommendation in location-based social networks. In *Privacy, Security, Risk and Trust (PASSAT), 2012 International Conference on and 2012 International Confernece on Social Computing (SocialCom)*, pages 144–153. IEEE, 2012.

[47] Femi Olumofin, Piotr K Tysowski, Ian Goldberg, and Urs Hengartner. Achieving efficient query privacy for location based services. In *Privacy Enhancing Technologies*, pages 93–110. Springer, 2010.

[48] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 514–523. ACM, 1990.

[49] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in cryptology-EUROCRYPT'99*, pages 223–238. Springer, 1999.

[50] Sarah Pidcock and Urs Hengartner. Zerosquare: A privacy-friendly location hub for geosocial applications. *Mobile Security Technologies*, page 83, 2013.

[51] Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In *Advances in Cryptology–CRYPTO 2010*, pages 502–519. Springer, 2010.

[52] Stephen C Pohlig and Martin E Hellman. An improved algorithm for computing logarithms over and its cryptographic significance (corresp.). *Information Theory, IEEE Transactions on*, 24(1):106–110, 1978.

[53] Tatiana Pontes, Marisa Vasconcelos, Jussara Almeida, Ponnurangam Kumaraguru, and Virgilio Almeida. We know where you live: privacy characterization of foursquare behavior. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, pages 898–905. ACM, 2012.

[54] Press release. Worldwide Smartphone Shipments Top One Billion Units for the First Time, according to idc. *http://www.idc.com/getdoc.jsp?containerId=prUS24645514*, 2014.

[55] Pierangela Samarati and Latanya Sweeney. Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression. Technical report, Technical report, SRI International, 1998.

[56] Salvatore Scellato, Anastasios Noulas, Renaud Lambiotte, and Cecilia Mascolo. Socio-spatial properties of online location-based social networks. *ICWSM*, 11:329–336, 2011.

[57] Elaine Shi, T-H Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((logN)^3)$ worst-case cost. In *Advances in Cryptology–ASIACRYPT 2011*, pages 197–214. Springer, 2011.

[58] Reza Shokri, Carmela Troncoso, Claudia Diaz, Julien Freudiger, and Jean-Pierre Hubaux. Unraveling an old cloak: k-anonymity for location privacy. In *Proceedings of the 9th annual ACM workshop on Privacy in the electronic society*, pages 115–118. ACM, 2010.

[59] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310. ACM, 2013.

[60] Peter Svensson. Smartphones now outsell 'dumb' phones. http://www.3news.co.nz/Smartphones-now-outsell-dumb-phones/tabid/412/articleID/295878/Default.aspx, 2013.

[61] Manolis Terrovitis. Privacy preservation in the dissemination of location data. *ACM SIGKDD Explorations Newsletter*, 13(1):6–18, 2011.

[62] Dylan Tweney. Google's location history web page shows all the places you've been, as logged by google maps. digital image. yes, google maps is tracking you. here's how to stop it. http://venturebeat.com/2014/08/17/yes-google-maps-is-tracking-you-heres-how-to-stop-it/, 2014.

[63] Carmen Ruiz Vicente, Dario Freni, Claudio Bettini, and Christian S Jensen. Location-related privacy in geo-social networks. *Internet Computing, IEEE*, 15(3):20–27, 2011.

[64] Peter Williams and Radu Sion. Usable PIR. In *NDSS*, 2008.

[65] Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 139–148. ACM, 2008.

[66] Yelp. http://www.yelp.com/, 2014.

[67] Tun-Hao You, Wen-Chih Peng, and Wang-Chien Lee. Protecting moving trajectories with dummies. In *Mobile Data Management, 2007 International Conference on*, pages 278–282. IEEE, 2007.

[68] Ge Zhong, Ian Goldberg, and Urs Hengartner. Louis, Lester and Pierre: Three protocols for location privacy. In *Privacy Enhancing Technologies*, pages 62–76. Springer, 2007.