Ph.D. Dissertation

Sensor Side-Channel Attacks on User Privacy: Analysis and Mitigation

Sashank Narain

College of Computer and Information Science

Northeastern University

Ph.D. Committee

Guevara Noubir	Advisor, Northeastern University
Kaushik Chowdhury	Northeastern University
Long Lu	Northeastern University
Igor Bilogrevic	External Member, Google Inc.

December 2017

Abstract

Mobile smartphones are equipped with an increasingly large number of precise and sophisticated sensors. These sensors vastly enhance the user's GUI experience, but they also raise the risk of directly or indirectly leaking their private information. Mobile operating systems (e.g., Android and iOS) mitigate such leakages by implementing app-level sandboxing and resource permissions. These protections may suffice for traditional privacy attacks using traditional hardware, however, they fail when attacks exploit side-channels that bypass the protections. One example of such side-channels is the motion sensors (Accelerometer, Gyroscope and Magnetometer) embedded in most modern smartphones. In this dissertation, we demonstrate two attacks that exploit the motion sensors on smartphones to infer accurate private information about the users such as their typed passwords and significant locations. To protect users from the above attacks and other location / sensor side-channel attacks, we propose the design and implementation of a mitigation framework called MATRIX for the Android ecosystem.

In the first part, we investigated the feasibility of keystroke inference when user taps on a soft keyboard of a smartphone are captured by the *Gyroscope* and *stereoscopic Microphones* sensors co-resident on the smartphone. Our experiments demonstrate that by building machine learning models specific to the keyboard, using a combination of multiple sensors and adequate filtering, it is possible to infer keystrokes with an accuracy of 90-94% on the standard Android QWERTY and Numeric keyboards.

In the second part, we investigated the feasibility of inferring a vehicular user's locations and traveled routes with high accuracy using information captured by the *Accelerometer*, *Gyroscope* and *Magnetometer* sensors co-resident on the smartphone. We modeled location inference as a maximum likelihood route identification problem on a graph generated from OpenStreetMap. Our simulations from 11 cities worldwide demonstrate that it is possible to output a ranked list of 10 routes containing the traveled route with

probability higher than 50%. We validate the simulations with over 980 km of real driving experiments from Boston and Waltham, MA that produce similar results.

In the third part, we discuss the design and implementation of the MATRIX framework built to protect users from location and sensor side-channel attacks. The MATRIX system gives users control and visibility over what and when location and sensors information is accessible to mobile apps. It implements a *PrivoScope* service that audits location and sensor accesses by all apps on the device and generates real-time notifications and graphs for visualizing these accesses; and a *Synthetic Location* service to enable users to provide obfuscated or synthetic location trajectories or sensor traces to apps they find useful, but do not trust with their private information. We also implemented a *Location Provider* component that generates realistic privacy-preserving synthetic identities and trajectories for users.

Acknowledgments

First of all, I would like to express my sincere gratitude to my Ph.D. advisor, Professor Guevara Noubir. He has been very supportive, patient and also extremely kind to me during this difficult phase of my life. He has not only helped me consolidate my ideas into research projects, but has also walked me through great technical depth whenever I needed assistance. Outside of work, his passion for research and commitment to helping students succeed in life has been a great source of inspiration for me. I consider myself an extremely lucky Ph.D. student to have been advised by him.

I am also very grateful to my colleagues and friends Amirali Sanatinia, Kenneth Block, Tien Vo. Huu, Triet Vo. Huu and others in my lab for supporting me, and giving me invaluable advice and insights. The discussions I had with them, the time spent, and the experiences they shared have helped me navigate many hurdles during this time. They have also significantly contributed to my personal development.

I would also like to sincerely thank Professor Kaushik Chowdhury, Professor Long Lu, and Dr. Igor Bilogrevic for accepting to be part of my Ph.D. committee. I really appreciate the invaluable time that you spent in evaluating my work and offering me very valuable feedback. Your feedback and comments have really helped me improve this work.

Finally, I would like to dedicate this dissertation to my parents Rameshwar and Manjula Narain, my brother Saurav, my sister-in-law Deepthi, my nephew Idhant, and my best friend and loving wife Tasneem. Their unconditional love and support have been my greatest strength during this journey. I am sincerely indebted to my brother Saurav for constantly motivating me to enroll in a Ph.D. program. He planted a seed in my mind years back that has now culminated into this dissertation. I am very grateful to Tasneem for spending long nights reviewing my work and discussing my research, which has helped me form new ideas that have culminated into this dissertation.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Mobile smartphones are presently the primary means for users globally to communicate, access information and even interact with the physical environment. They are used for various day-to-day and business activities where several of those activities deal with sensitive information like bank credentials, credit card numbers, email passwords, health records, and location information. These devices are equipped with an increasingly large number of precise and sophisticated sensors. These sensors vastly improve the quality of the user's interaction with the environment. To illustrate an example, the addition of high accuracy GPS chips and Magnetometers have enabled the development of navigation apps like Google Maps and Waze easing user commutes. Similarly, the addition of Accelerometers and Gyroscopes have enabled the development of high quality user-interactive games for the mobile platform.

1.1 Privacy Leakage from Smartphone Sensors

Smartphone sensors also pose significant threats for privacy breaches as they directly or indirectly leak private information about their users. The leakage of location information from the GPS sensor, for instance, has been a fast growing privacy concern. The commercial GPS hardware available in modern smartphones is capable of triangulating a user's position within an accuracy of 3 meters. This GPS data can be fused with Wi-Fi and cellular data to provide high accuracy location data even indoors, where GPS is ineffective. This leakage enables more sophisticated threats such as tracking users, identity discovery, and identification of home and work locations. It also increases the risk of the discovery of behaviors, habits, preferences and one's social network, which can potentially lead to effective physical and targeted social engineering. To illustrate an example, the 'Brightest Flashlight' app on the Google Play Store was recently sued by the Federal Trade Commis-

sion (FTC) for deceiving customers and sharing their location information without their knowledge [1]. This app with 4.7 stars rating and over one million users is just one example of seemingly innocuous applications that deceive users. Despite the protections put in place by mobile operating systems, applications can access sensitive information in ways difficult for users to control and fully grasp. For example, when granted a permission to access location information, an Android app can track users even after they close the app and even after the phone is restarted.

The leakage of location information from the GPS sensor can be detected by careful users who monitor the permissions of the apps they install and use on their device. For example, a careful user can choose to not install a Flashlight app that requests location permissions. A harder problem is to protect privacy leakage from side channel attacks exploiting zero-permission sensors such as the Accelerometers, Gyroscopes and Magnetometers. Access to these sensors on current mobile operating systems (e.g., Android and iOS) does not require any permissions from the users, nor does the system show any notifications to the users[1]. This expanding attack surface is an attractive target for those seeking to stealthily exploit privacy information, especially as users become increasingly aware of location tracking systems [2, 3] and attempt to minimize their exposure by disabling, limiting usage of, or removing tracking apps.

1.2 High-Level Overview

In this dissertation, we demonstrate two attacks that exploit the motion sensors on smartphones to infer accurate private information about the users such as their typed passwords and significant locations. In the first attack, we investigated the feasibility of keystroke inference when user taps on a soft keyboard of a smartphone are captured by the *Gyroscope* and *stereoscopic Microphones* sensors co-resident on the smartphone. Our experiments demonstrate that by building machine learning models specific to the keyboard, using a combination of multiple sensors and adequate filtering, it is possible to infer keystrokes with an accuracy of 90-94% on the standard Android QWERTY and Nu-

[1] As of now (Android 8.1) access to Accelerometer, Gyroscope, and Magnetometer is automatically granted without user warnings nor explicit permission request in the manifest file.

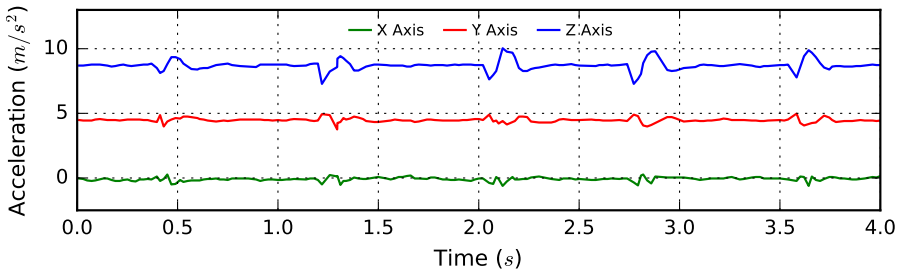
meric keyboards. In the second attack, we investigated the feasibility of inferring a vehicular user's locations and traveled routes with high accuracy using information captured by the *Accelerometer*, *Gyroscope* and *Magnetometer* sensors co-resident on the smartphone. We modeled location inference as a maximum likelihood route identification problem on a graph generated from OpenStreetMap [4]. Our simulations from 11 cities worldwide demonstrate that it is possible to output a ranked list of 10 routes containing the traveled route with probability higher than 50%. We validate the simulations with over 980 km of real driving experiments from Boston and Waltham, MA that produce similar results.

To protect users from the above attacks and other sensor side-channel / location attacks, we designed and implemented a mitigation framework called MATRIX for the Android ecosystem. The MATRIX system gives users control and visibility over what and when location and sensors information is accessible to mobile apps. It implements two services: a *PrivoScope* service that audits location and sensor accesses by all apps on the device and generates real-time notifications and graphs for visualizing these accesses; and a *Synthetic Location* service to enable users to provide obfuscated or synthetic location trajectories or sensor traces to apps they find useful, but do not trust with their private information. We also implemented a *Location Provider* component that generates realistic privacy-preserving synthetic trajectories by modeling user locations' and their movements as Finite State Machines (FSM) with probabilistic transitions connecting states, user schedule as a randomized linear program, incorporating traffic information into routes from historical traffic APIs such as Google Maps Directions API [5], and shaping them using statistical information about speed and acceleration from users behavior.

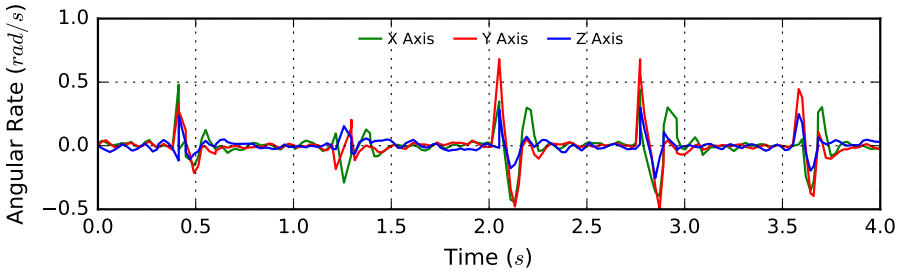
1.3 Single-stroke Language-Agnostic Keylogging

1.3.1 Attack Opportunity

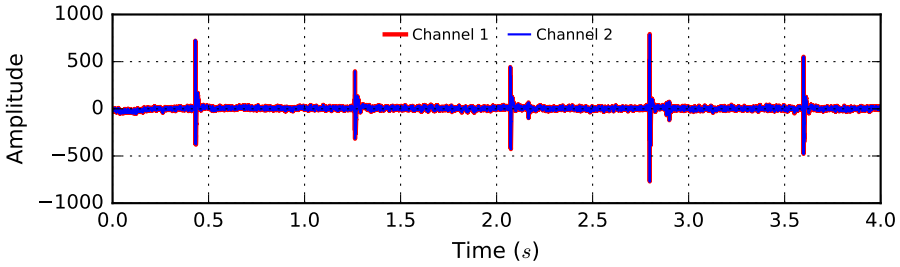
The Accelerometer and Gyroscope sensors embedded in most modern smartphones can detect keystroke vibrations when a user types on a soft keyboard. The magnitude and orientation of these vibrations vary depending on the tap location, however, they are quite similar for the same location. These vibrations can be mapped to the standard



(a) Recorded Accelerometer vibrations



(b) Recorded Gyroscope vibrations

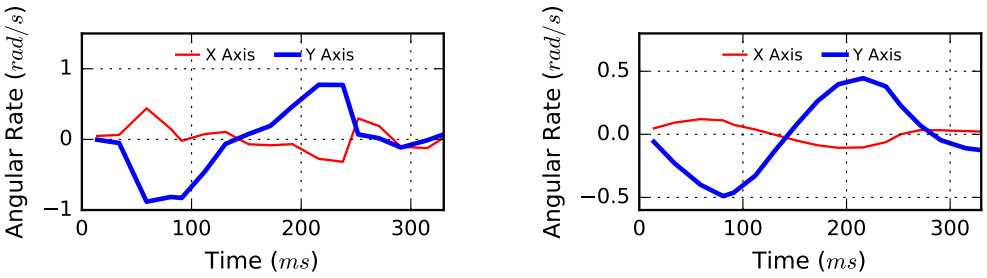


(c) Recorded Microphone taps

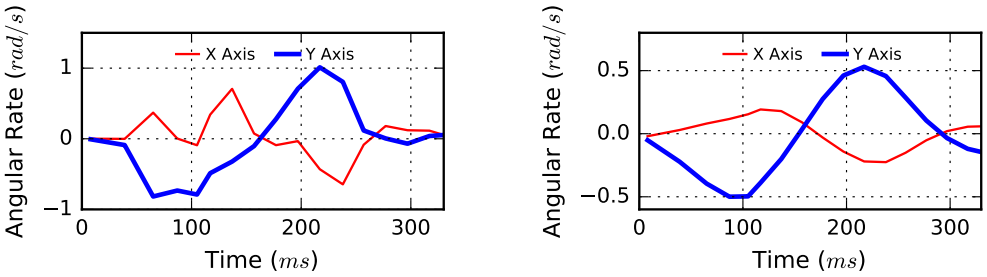
Figure 1.1: An example of the Accelerometer and Gyroscope vibrations and Microphone tap sounds for the sequence 'HELLO'.

QWERTY and Number keyboard layout creating a potential keylogger that infers user keystrokes based on the vibration patterns. Microphone arrays are also becoming increasing commonplace in smartphones. These microphones are sensitive enough to record user keystroke tap sounds on a soft keyboard. The amplitude of the audio signal captured by different microphones and the time delay between signals reaching the microphones can also be analyzed for creating a potential keylogger.

Figure 1.1 shows the vibrations recorded by the Accelerometer (Figure 1.1a) and Gy-



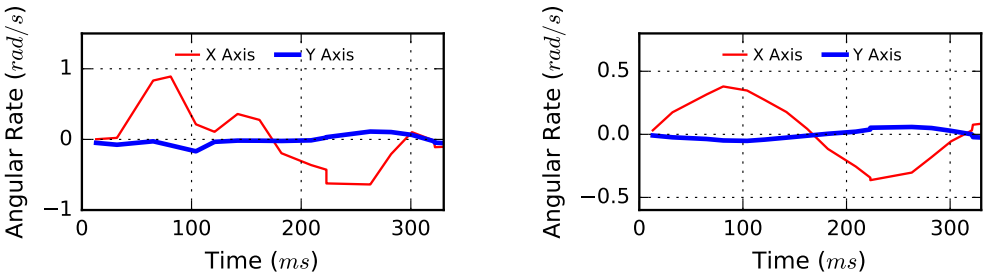
(a) Raw keystroke (left) and filtered keystroke (right) for first keystroke 'Q'



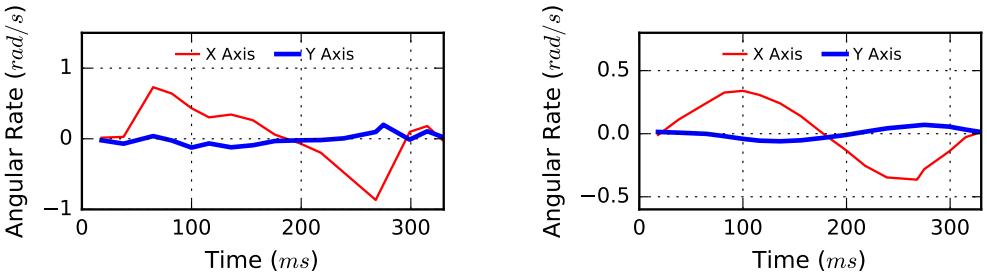
(b) Raw keystroke (left) and filtered keystroke (right) for second keystroke 'Q'

Figure 1.2: An example of two raw Gyroscope keystrokes and the corresponding filtered keystrokes for the letter 'Q'.

roscope (Figure 1.1b) and tap sounds recorded by the Microphones (Figure 1.1c) on a HTC One smartphone for the sequence 'HELLO'. Note that the amplitude of the keystroke vibrations and sounds are significantly higher than at rest between two keystrokes. Also, note that Accelerometer and Gyroscope vibrations for two keystrokes 'L' are similar to each other and significantly different from the keystroke 'H' and 'E'. The keystrokes for 'L' and 'O' look similar as they are adjacent keys on the QWERTY keyboard. Figures 1.2 and 1.3 show the raw and corresponding filtered Gyroscope data for two keystrokes each for the letters 'Q' and 'V', respectively. Note that the raw Gyroscope data is affected by noise in all the cases, that can be removed by frequency filtering. The filtered data for two keystrokes of the letter 'Q' are similar to each other. Similarly, the filtered data for two keystrokes of the letter 'V' are similar to each other but very distinct from those for the letter 'Q'. This implies that the attack accuracy can substantially benefit from noise reduction using frequency filtering techniques.



(a) Raw keystroke (left) and filtered keystroke (right) for first keystroke 'V'



(b) Raw keystroke (left) and filtered keystroke (right) for second keystroke 'V'

Figure 1.3: An example of two raw Gyroscope keystrokes and the corresponding filtered keystrokes for the letter 'V'.

1.3.2 Motivation

The topic of keystroke inference from smartphone sensors has been studied by numerous research groups in recent years. All of these studies used larger keyboard sizes and different device orientations to infer user keystrokes. Cai & Chen [6] attempted to infer number keystrokes on a Number only keyboard in Landscape mode using the Orientation sensor. They collected three data-sets on a HTC Evo 4G smartphone and achieved a successful inference accuracy of about 70% on all three data-sets. Xu, Bai & Zhu [7] attempted to infer lock screen passwords and numbers entered during a phone call using the Accelerometer and Orientation sensor. They collected data-sets of several tap events from three students using two smartphones, HTC Aria and Google Nexus (One), and achieved an accuracy of about 99% for one user on the Google Nexus (One) and about 70% - 85% for the other users. Aviv et al. [8] attempted to infer PIN and pattern passwords using the Accelerometer on four different smartphones; Nexus One, G2, Nexus S

and Droid Incredible. They reached an accuracy of 43% and 73% for PIN and pattern passwords respectively, within 5 attempts from a set of 50 PINs and 50 patterns in a controlled setting. Owusu et al. [9] attempted to infer characters on a QWERTY keyboard in Landscape mode using the Accelerometer. They collected several data-sets on a HTC ADR 6300 smartphone from four participants and showed that, out of 99 6-character passwords, it was possible to successfully infer 6 character passwords in 5 trials. Miluzzo et al. [10] attempted to infer characters on a QWERTY keyboard in Landscape mode and icon locations in Portrait mode using the Accelerometer and Gyroscope. They collected a data-set on the Google Nexus S, Samsung Galaxy Tab 10.1 and iPhone 4 and showed that locations of icons can be inferred with 79% and 65% accuracy for the iPhone and Google Nexus S respectively, and characters could be inferred with 65% accuracy. Other sensor attack vectors include the camera to create a 3D model of the user's environment to zoom in objects of interest [11], the microphone to steal private information (e.g., credit card numbers) from phone conversations, the Accelerometer to infer smartwatch keystrokes [12], and the Accelerometer to steal text typed on a physical keyboard in close proximity to the smartphone [13].

The above attacks focus on keyboards that represent larger key sizes than the standard Android QWERTY and Number keyboard. Our motivation for this work is to demonstrate that by using a combination of acoustics and sensors and a multi-tier approach based on the areas of keyboards, one can achieve a high prediction accuracy even on the standard Android QWERTY and Number keyboard. Another motivation is to demonstrate the feasibility of character and number inference using the sounds generated by the keystrokes and recorded by a device's stereoscopic microphones. To achieve our motivation, we developed an algorithm and framework based on statistical methods and machine learning that can predict keystrokes without repetition or multiple attempts. Our framework is language agnostic as we do not use any lexical properties of languages, however, we do assume that the adversary knows the keyboard layout. We demonstrate the algorithm using data collected at an office and in a restaurant. A malicious application and a weak permission model for Android sensors coupled with data modeling techniques make our attack feasible and consequential. We also show that the audio data

can be combined with the Gyroscope to further boost the inference accuracy.

1.3.3 Contributions

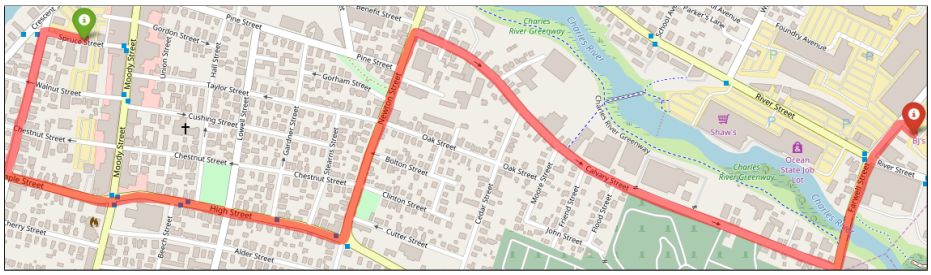
Our contributions can be summarized as follows:

- We demonstrate that by recording the keystroke vibrations from the Gyroscope sensor and the tap sounds from the stereoscopic Microphones co-resident on a smartphone, it is possible to infer user typed keys with a reasonably high accuracy of above 90%. We also show that this accuracy can be boosted by combining the Gyroscope data and the Audio data. We implemented a system that can process this raw keystroke data, perform noise filtering, build training models and use these models to make language agnostic keystroke predictions on unknown test data.
- We designed and implemented a specialized meta-algorithm that divides the keyboard into areas and trains models using the Gyroscope and the Audio data specific to those areas. The algorithm combines character-specific and area-specific models to make more accurate predictions. We show that by combining our algorithm with meta-algorithms such as Bagging and Boosting [14], we were able to achieve higher per algorithm accuracy than an elementary use of the machine learning algorithms on unknown test data.
- We also demonstrate the feasibility of Trojan apps that periodically query the Android operating system for the foreground activity and can covertly record the Gyroscope and Microphones when a sensitive activity (e.g., Bank app) is being used.

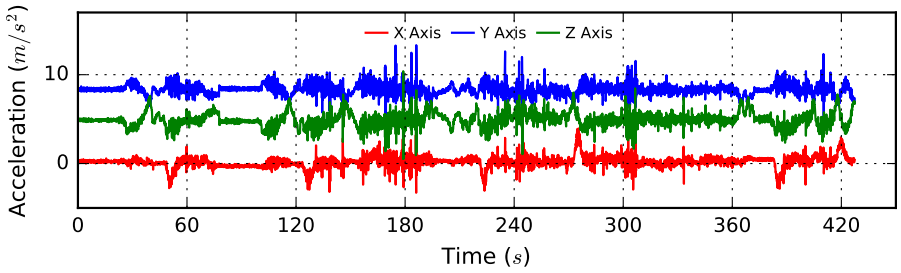
1.4 Inferring User Routes and Locations

1.4.1 Attack Opportunity

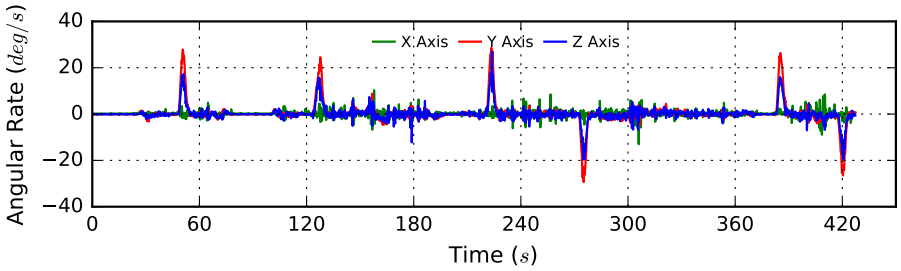
The Gyroscope, Accelerometer and Magnetometer sensors on a smartphone can provide useful information about a user's route. Gyroscopes report the rate of angular change experienced by a smartphone in the x, y and z axes which can be used to estimate turn and curvature information. Accelerometers report the accelerations experienced by the smartphone in all axes which translates to the speed of the vehicle. Magnetometers report



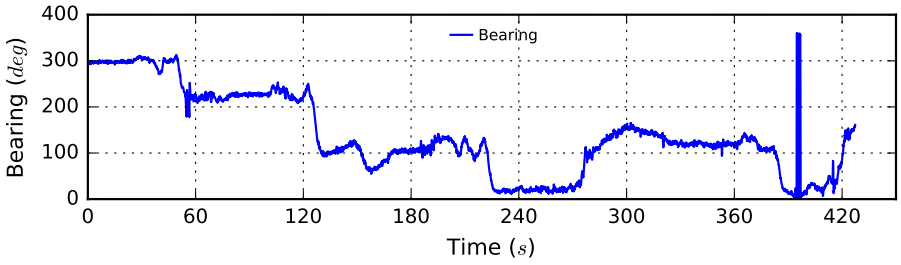
(a) Experimental route



(b) Recorded Accelerometer data



(c) Recorded Gyroscope data



(d) Recorded Magnetometer data

Figure 1.4: An experimental route and the recorded Accelerometer, Gyroscope and Magnetometer data.

the position of the smartphone with respect to magnetic North and can be used to estimate the direction of heading of the vehicle. The information from these sensors can be combined to form a route's attributes which is then potentially traceable on a public map resource. This can become an effective attack vector for tracking the vehicular movements of users which can, in turn, easily lead to inferring the home and workplace of the user. Typically, Accelerometers and Magnetometers are very unreliable due to sensitivity to environmental factors but can still be used to calculate idle times and heading directions.

Figure 1.4 shows an example experimental driving route (Figure 1.4a) and the recorded Accelerometer (Figure 1.4b), Gyroscope (Figure 1.4c) and Magnetometer (Figure 1.4d) values from the smartphone. Note that the Gyroscope sensor reports significant positive deviations for every left turn and negative deviations for every right turn, while the Magnetometer reports significant heading changes with respect to magnetic North for these turns. The Gyroscope deviations can be integrated over time to calculate the turn angles and curvatures of this route. The smaller deviations recorded by the two sensors can be attributed to both road curvature and sensor / environmental noise. Also note that the Accelerometer sensor reports the vehicle's accelerations at all moments of the route. These accelerations also contain sensor / environmental noise caused from vehicle vibrations, road slopes and irregularities. The large spikes on the z axis can typically be attributed to the quality of the road (e.g., potholes).

1.4.2 Motivation

The topic of location privacy has been extensively studied since the early days of mobile phones. Cellular communication systems, as early as GSM, *attempted* to protect users' identity by using temporary identifiers (e.g., TMSI) that increased the difficulty of tracking users. The attack surface significantly expanded recently with the pervasiveness of mobile and sensing devices, open mobile platforms (running untrusted code) and ubiquitous connectivity. One such example of location tracking involves monitoring the MAC address of probe packets periodically transmitted by Wi-Fi cards. This has been exploited by marketing companies and location analytics firms. For example, companies such as Euclid Analytics state on their website that they collect "the presence of the device, its sig-

nal strength, its manufacturer, and a unique identifier known as its Media Access Control (MAC) address" in shopping malls [15]. This information is used to analyze large spatio-temporal user traffic patterns. Another example is the startup Renew, that installed a large number of recycling bins in London with the capability to track users. This allowed them to identify users and their specific route and walking speed [16, 17]. The practicality of such attacks using the physical and link layer information, however, remains limited to adversaries with a physical presence in the vicinity of the user or with access to the ISP infrastructure. Also, these attacks can be defeated by MAC address randomization schemes such as the one introduced by Apple in its iOS 8 release [18].

Attacks that exploit the open nature of mobile platforms are more concerning as they can be remotely triggered (e.g., from distant countries beyond the jurisdiction of a victim's country's courts of law), and require virtually no deployment of physical infrastructure. One common example of a location tracking attack is a malicious app obtaining and exfiltrating the user's location by accessing the mobile device's location services that typically rely on GPS, Wi-Fi, or cellular signals. Such attacks can be detected because the app must request permission to access the location services on the smartphone. The attack can also be defeated by users who disable location access for the app. Recently, researchers have demonstrated stealthier attack vectors that rely entirely on zero-permission resources. For example, Michalevsky et al. [19] leveraged the power usage of the phone and matched them with known location power profiles to infer location. This technique is unreliable when installed apps (e.g., Facebook, Skype) on the phone drain power. Zhou et al. [20] analyzed speaker on/off times controlled by a GPS based navigation app to infer which course a driver took based on the duration of audible driving instructions. This technique is unreliable in situations when a user is stuck in traffic and also relies entirely on the user using a navigation app while driving. Han et al. [21] exploited the Accelerometer and Magnetometer, and leveraged a probabilistic dead reckoning method called Probabilistic Inertial Navigation (ProbIN) to map probability of displacement to probability of motion. They trained models using an association of sensor data and map data and observed a resolution approaching 200 meters for their test data. This attack is not scalable due to the need for acquiring sensor data corresponding to map data to train the models.

Nawaz et al. [22] demonstrated that an Accelerometer and Gyroscope can be used to identify 'significant' journeys independent of phone orientation and traffic. They applied Dynamic Time Warping (DTW) to calculate the distance between various journeys and use a k-medoids clustering approach to cluster similar routes together. Their attack is not scalable because it uses the DTW algorithm that requires significant computing power and resources for multiple routes.

The above attacks are impractical for large scale location tracking as they rely on some historical information to make location inferences. Our motivation for this work is to demonstrate the potential of tracking user mobility solely using the zero-permission Accelerometer, Gyroscope and Magnetometer sensors on the smartphone, without explicitly requesting permissions to access the location services, and without any prior knowledge about the user. To achieve our motivation, we model a user trajectory as a route on a graph $G = (V, E)$, where the vertices represent road segments and the edges represent intersections. We formulate the identification of a user trajectory as the problem of finding the maximum likelihood route on G given the sensors' samples. Using techniques similar to trellis codes decoding, we developed an algorithm that identifies the most likely routes by minimizing a route scoring metric. The knowledge of the user's route can easily lead to inferring the home and workplace of the user. Further information about the user's identity can be derived by inspecting the town's public database.

1.4.3 Contributions

Our contributions can be summarized as follows:

- We demonstrate that location tracking using zero-permission smartphone sensors can be modeled as a graph-theoretic problem. We also demonstrate that user trajectory inference from sensor data can be formulated as a problem of finding the maximum likelihood route on the graph. To efficiently search the sensor data on the graph, we designed and implemented an efficient location/trajectory inference algorithm that incorporates road segments curvature, travel time, turn angles, Magnetometer information, and speed limits to identify the most likely routes, by minimizing a route scoring metric.

- We designed and implemented a location tracking framework to assess the potential of this attack in realistic environments. The framework consists of six building blocks: (1) road graph construction from the OpenStreetMap map data, (2) filtering and processing the sensor data, and generating a compact sequence of tags that match the semantic of a graph route, (3) a maximum likelihood route identification algorithm, (4) simulation tool for generating realistic routes, (5) a mobile app to record sensor data from real experiments, and (6) trajectory inference for simulated and real mobility traces.
- We performed extensive simulations for 11 cities around the world with varying population, road densities and topologies (including Atlanta, Boston, London, Manhattan, Paris and Rome). The results demonstrate that for most cities, it is possible to output a short list of 10 routes containing the traveled route with probability higher than 50%.
- We collected real driving measurements in Boston and Waltham, Massachusetts, spanning over 980 km. The results demonstrate a probability of 30% (resp. 60%) of inferring a list of 10 routes containing the true route in Boston (resp. Waltham). These results are similar to simulations which implies that our simulation framework may serve as an effective model for studying the attack in a larger scale where experiments are limited.

1.5 Mitigating Location Leakage with Dynamic App Sandboxing

1.5.1 Motivation

The protections against location tracking attacks mostly revolve around obfuscating the users' location. Several research works have proposed solutions that induce noise in the location data such that a user's actual location cannot be derived from the resultant location [23, 24, 25, 26, 27]. Others have devised solutions that sends the real location with several dummy locations or within a data-set, and uses the query response pertaining to the real location while discarding the rest [28, 29, 30, 31, 32, 33]. Others have proposed stripping off all identifying information about a user before sending the real location

data in order to protect the user's privacy [34, 35]. Unfortunately, these solutions still leak some information about their users and can be combined with other data (e.g., census data) to infer user identities and their locations [36, 37]. Moreover, incomplete or incorrect implementations of these solutions make them vulnerable to location discovery attacks.

Other protections against location tracking attacks include but are not limited to, recommending new security frameworks [2, 38, 39, 40, 41, 42, 43, 44, 45], tainting sensitive data [46, 47], dynamic analysis [48, 49], static code analysis [50, 51, 52, 53], permissions analysis [54], application retrofitting [55, 56, 57], analyzing Internet traffic for sensitive information [58, 59], and even cryptographic techniques [60]. Most of these works are orthogonal to the system we propose as their motivation and techniques differ. Mobile operating systems also try to prevent undesired location tracking by implementing permissions that all apps must request for accessing location data. These measures, however, are not very effective in preventing location tracking as users are often careless about granting such permissions.

An alternative protection against location tracking attacks is the generation of synthetic location trajectories [61, 62] which are independent of users real locations [63, 64]. These trajectories guarantee location privacy because it is not possible to derive the user's location from them, however, they risk denial of service if an adversary detects that the trajectories are fake. To be effective against such detection, these trajectories must emulate real movements and routes by incorporating real user transitions, movement schedules, driving behavior and traffic information.

1.5.2 Proposed Solution

The proposed MATRIX framework generates realistic privacy preserving synthetic mobility trajectories that are difficult to distinguish from real ones by an adversary. These synthetic trajectories can also be customized by the users according to their preferences. MATRIX dynamically and seamlessly sandboxes apps installed on the smartphone to receive synthetic feeds if specified by the user. To this end, we model user locations' and their movements from one location to another through Finite State Machines (FSM) with probabilistic transitions connecting states. The states can represent being at home, at

work, or being at a restaurant. The transitions between states represent routes that are generated from graphs constructed from real road networks. To make the synthetic trajectory realistic to an adversary, we first generate a randomized schedule (path in the FSM) that satisfies the preferences in terms of timing, and expected presence in specific states. This problem of generating a schedule is formulated as a randomized linear program. We further incorporate traffic information from historical traffic APIs such as Google Maps Directions API, shape them using statistical information about speed and acceleration from users behavior, and also add noise to the synthetic data to emulate real GPS data, in addition to incorporating walk times and idle times. One key aspect of MATRIX is that it is a framework that was implemented and evaluated on Android mobile devices, and features other capabilities such as seamlessly injecting synthetic sensor feeds, and a PrivoScope service to monitor and analyze apps patterns for accessing location and sensor services. PrivoScope provides users with real-time notifications and a graphical interface to show how apps access their location information, or even permissionless sensors (e.g., the time of location access, the accuracy of the location data received, the rate a sensor was sampled, and whether the app was in foreground or background). This helps users make more privacy informed decisions about providing synthetic location data to apps using the MATRIX framework or uninstalling/disabling apps they do not trust. Finally, MATRIX can be used by security researchers to identify which apps misuse / leak private location and sensors information, by injecting synthetic honey-data and observing if it gets used in contexts not authorized by the users.

1.5.3 Contributions

Our contributions can be summarized as follows:

- We designed and implemented MATRIX, a framework and system for automatically generating realistic privacy preserving synthetic identities and mobility trajectories for users. The system generates these profiles by modeling user movements and incorporating historical traffic and user driving characteristics to the movements. MATRIX is integrated within Android without modifications to the operating system, and allows a user to specify which apps should seamlessly receive real feeds

and when, and which apps should receive synthetic feeds.

- We show that the basic problem of generating random mobility patterns with spatio-temporal constraints can be formulated as a randomized Linear Program. To mitigate detection, MATRIX derives random routes with historical traffic information and incorporates users speed and acceleration characteristics.
- We designed and implemented a PrivoScope service within MATRIX that enables users to monitor, visualize, audit, and analyze when and how an app accesses their location data and phone sensors, and potentially any sensitive service. The PrivoScope service also exposes a permission-protected API that allows other security apps installed on the smartphone to get real-time information about which apps access private location and sensors information.
- We extensively evaluated MATRIX to validate performance, reliability, and verify realism of synthetic trajectories. On a set of 1000 popular Android apps, we report negligible impact in terms of performance and reliability. On a set of 10 popular location-driven apps, we report that MATRIX is undetected while at least one app could detect fake non-MATRIX mobility patterns. Our user study involving 100 users indicates that the synthetic trajectories generated are difficult to differentiate from real traces visually, with more users confusing synthetic trajectories to be real. Our machine learning classification indicates that even a well trained algorithm only achieves 63% accuracy correctly guessing if a trajectory is synthetic (compared to 50% for an algorithm that uses a coin-flip).

1.6 Outline

The rest of the dissertation is structured as follows. In Chapter 2, we demonstrate the keylogging attack that uses the *Gyroscope* and *stereoscopic Microphones* on a smartphone, combined with filtering algorithms and domain-specific machine learning to infer keystrokes typed by a user. In Chapter 3, we demonstrate the location tracking attack that uses the *Accelerometer*, *Gyroscope* and *Magnetometer* on a smartphone, combined with filtering algorithms and efficient search algorithms on real map data to infer user traveled vehicular

routes and locations with high accuracy. In Chapter 4, we describe the design and implementation of the MATRIX framework that addresses some current privacy protection weaknesses in the Android ecosystem and provides users with a tool to analyze how apps access their private information as well as the capability to change how certain untrusted apps receive location and sensor data. In Chapter 5, we discuss some of the future directions of this work.

Chapter 2

Single-Stroke Language-Agnostic Keylogging

This chapter describes our keystroke inference system that extracts recordings of keystrokes from the Gyroscope and Microphone sensors, performs noise filtering on the data, consolidates the data, trains machine learning algorithms using known keystrokes, and uses the models to infer unknown keystrokes. In Section 2.1, we describe a scenario of how a stealthy attack can be launched against a sensitive Android app by a victim inadvertently downloading a malicious app. In Section 2.2, we describe the high-level architecture of our system and discuss why the Gyroscope and Microphone sensors leak information about a user's typing activity. In Section 2.3, we describe our automated keystroke inference system, data collection process and the meta-algorithm. In Section 2.4, we outline the evaluation metrics and present the results of our evaluations. In Section 2.5, we describe previous related work and we conclude in Section 2.6.

2.1 Attack Vector

The adversary follows the steps described below to perform a successful attack. They develop and distribute a malicious app usually as a Trojan and trick the victim into installing the app through techniques such as social engineering (e.g., a malicious app disguised as a game or a note taking app). Previous works have found examples of such apps with backdoors in the Android marketplace [65, 66]. This app performs two roles after installation on the victim's smartphone. First, it presents a custom keyboard to the user to collect typing behavior for training models. Second, after training, it listens in the background for keystrokes from sensitive Android applications.

To successfully perform the first role, the malicious app starts collecting user's typing behavior by using a custom keyboard and recording the Gyroscope and stereo Microphones. The microphone requires permissions and the adversary needs to declare it

in the *manifest* file. To avoid raising the victim's suspicions, the adversary justifies such permissions by providing functionality such as: note taking and voice recognition. The Trojan uploads this collected training data to a remote server to build prediction models specific to the victim. The adversary can also use generic training data and prediction models as a trade-off for performance and stealth.

To accomplish the second role, the Trojan runs in the background and queries the operating system for the current foreground app. To reduce battery drain, these queries run at predefined conservative intervals. On inferring that the victim opened a sensitive app (e.g., a bank app), the Trojan starts to actively collect the Gyroscope and Microphone data. The current foreground app can be found by using the `ActivityManager` class in Android SDK. The recorded data is filtered and keystrokes are extracted by the application and evaluated using the training models to infer the user's typed keystrokes.

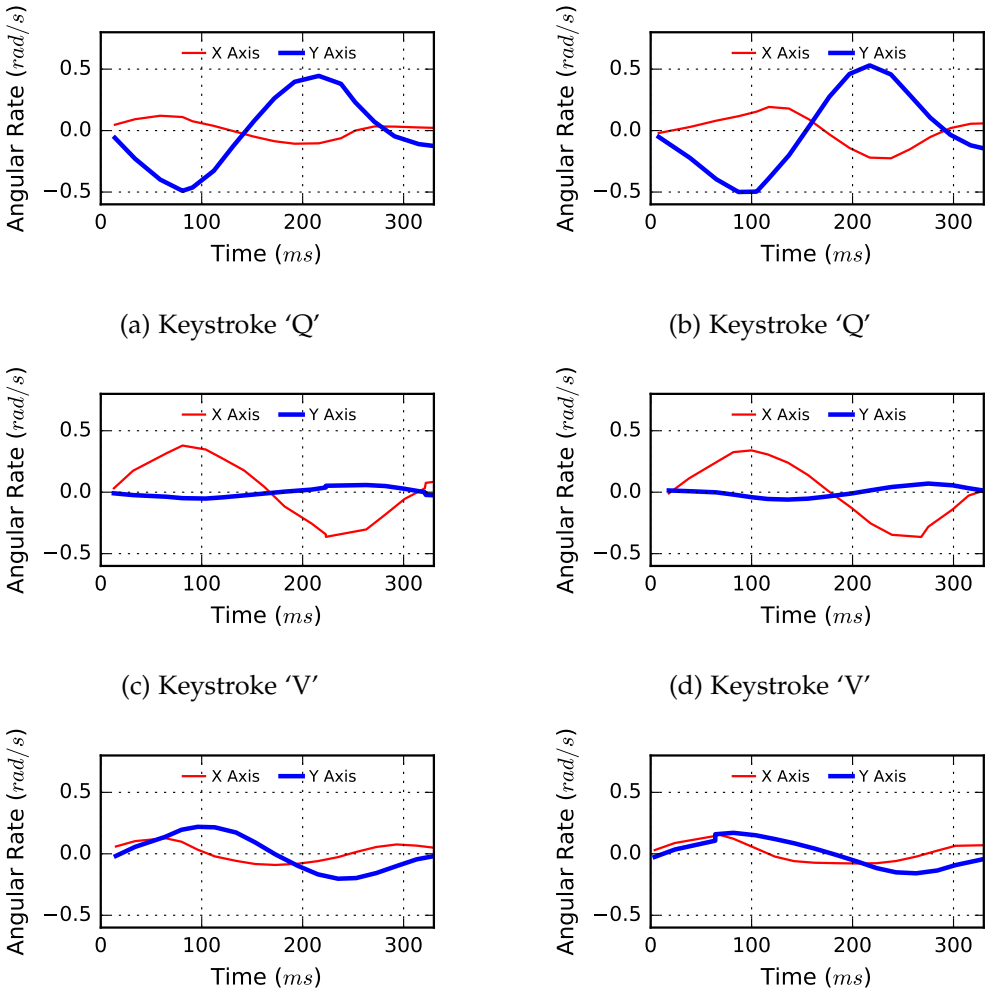
2.2 High-Level Approach

In this section, we describe the architecture of our keystroke inference system and also discuss why the Gyroscope and Microphone sensors leak information about a user's typing activity.

2.2.1 Keystroke Inference from Sensor Data

Gyroscopes: The magnitude and orientation of the Gyroscope vibrations for every user keystroke varies depending on the keystroke location. It also varies from device to device as the vibrations reported are dependent on the location of the Gyroscope hardware. These vibrations can be mapped to a standard fixed keyboard layout. Figure 2.2 shows the location of the Gyroscope (in red), the Android sensor coordinate system relative to the Gyroscope hardware, and the location of keys 'Q', 'V' and 'I' on a standard QWERTY keyboard on the HTC One. Figure 2.1 shows the vibrations experienced by two keystrokes each of keys 'Q', 'V' and 'I'. We see that key 'Q' shows significant vibration in the y axis and key 'V' shows significant vibration in the x axis. As the key 'I' is close to both the axes, it does not show considerable vibration in both the axes.

The Gyroscope sensor can be easily accessed using the Android SDK [67] APIs de-



(a) Keystroke 'Q'

(b) Keystroke 'Q'

(c) Keystroke 'V'

(d) Keystroke 'V'

(e) Keystroke 'I'

(f) Keystroke 'I'

Figure 2.1: Similarity between two keystrokes each for letters 'Q', 'V' and 'I' on a standard QWERTY keyboard on the HTC One.

fined by the `SensorManager` class. A potential issue with using the SDK API is that the sampling rate is not fixed and may reduce when more processor intensive services are running, thereby reducing the inference accuracy. A solution for obtaining high and constant sampling rate on a Android smartphone even when other high priority services are running on the system is to use the Android NDK [68] APIs defined in `sensors.h`.

Microphones: Microphone arrays are becoming commonplace in smartphones. These microphones are sensitive enough to capture user keystroke sounds on the soft keyboard.

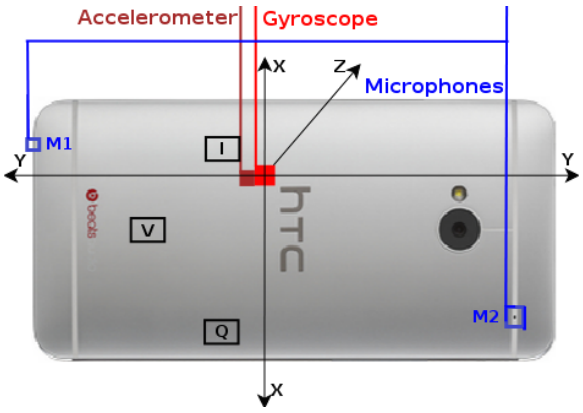


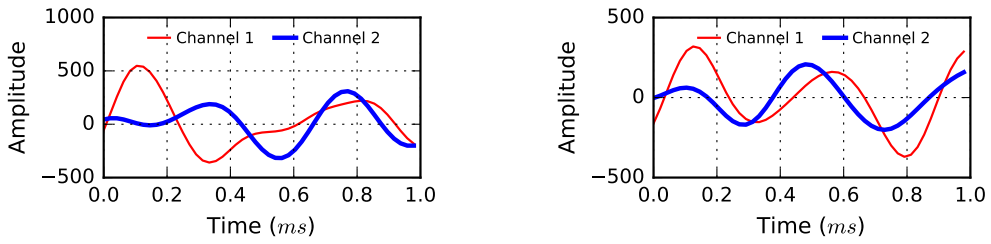
Figure 2.2: Location of the Accelerometer, Gyroscope and Microphones on the HTC One; Approximate location of keys 'I', 'Q' and 'V' on the standard QWERTY keyboard.

Most newer devices (e.g., Apple iPhones, Google Pixel, HTC One series) are equipped with microphone arrays that support stereo recording. Many other smartphones are equipped with dual membrane microphones that focus on capturing different sound levels (one for sensitivity and the other for distance) on a single microphone. In both these arrangements, the audio captured by the arrays are combined and processed to provide high quality and distortion free audio recording to users with the capability to detect feeble sounds. This yields two attack vectors for inferring keystrokes: one that uses the amplitude of audio signals at the microphones and the other that uses the time delay between signals reaching the two microphones. We use a combination of both the techniques to build our inference models.

Figure 2.2 shows the location of the Microphones (in blue) on the HTC One. For a user tap location L on a device with maximum sampling rate T_s, the sampling delay T_L between two microphones L_{M1} and L_{M2} can be calculated as:

$$T_L = \frac{T_s * |d(L, L_{M1}) - d(L, L_{M2})|}{c}$$

Here, $d(L, L_{M1})$ is the distance between the tap location L and microphone L_{M1}, $d(L, L_{M2})$ is the distance between the tap location L and microphone L_{M2}, and c is the speed of sound (340 m/s). The distance between the two microphones on the HTC One is about 0.134 m. The current maximum supported sampling rate for Android is 48 KHz.



(a) Keystroke 'Q'

(b) Keystroke 'V'

Figure 2.3: Sound waves of keystrokes recorded by the HTC One for keys 'Q' and 'V'.

Using these values with the formula, a difference of 18 samples is obtained for taps in close proximity to the microphones. This means that taps for different keystrokes will produce varying sample differences based on their distance from the two microphones. This difference will increase when smartphones start supporting higher sampling rates such as 192 KHz, currently supported by Blue-ray. Using a rate of 192 KHz with the formula, a difference of 75 samples can be obtained for taps in close proximity to the microphones. This will significantly improve the accuracy of inference and is indicative of the impact of the sampling rate on the accuracy. To illustrate the time delay between two microphones, we recorded multiple keystrokes for keys 'Q' and 'V' on a standard QWERTY keyboard on the HTC One at the maximum sampling rate of 48 KHz. Figure 2.3 shows an example of a single tap for the two keys. We found that, for multiple taps, the two keys always have a delay between $8-10$ and $15-17$, respectively.

The Microphones can be easily accessed using the Android SDK APIs defined by the `AudioRecord` class. Even though they require permissions from the user, they can run in the background after the user accepts the permission without showing a notification. We believe that this capability can be used maliciously and should be addressed.

2.2.2 System Architecture

The architecture of our keystroke inference system is composed of the components shown in Figure 2.4. The *App* component is the Trojan that secretly records the Gyroscope and stereo Microphones when a user types in our app (training data) or another security sensitive app (test data). The raw Gyroscope and Audio data is uploaded to a colluding

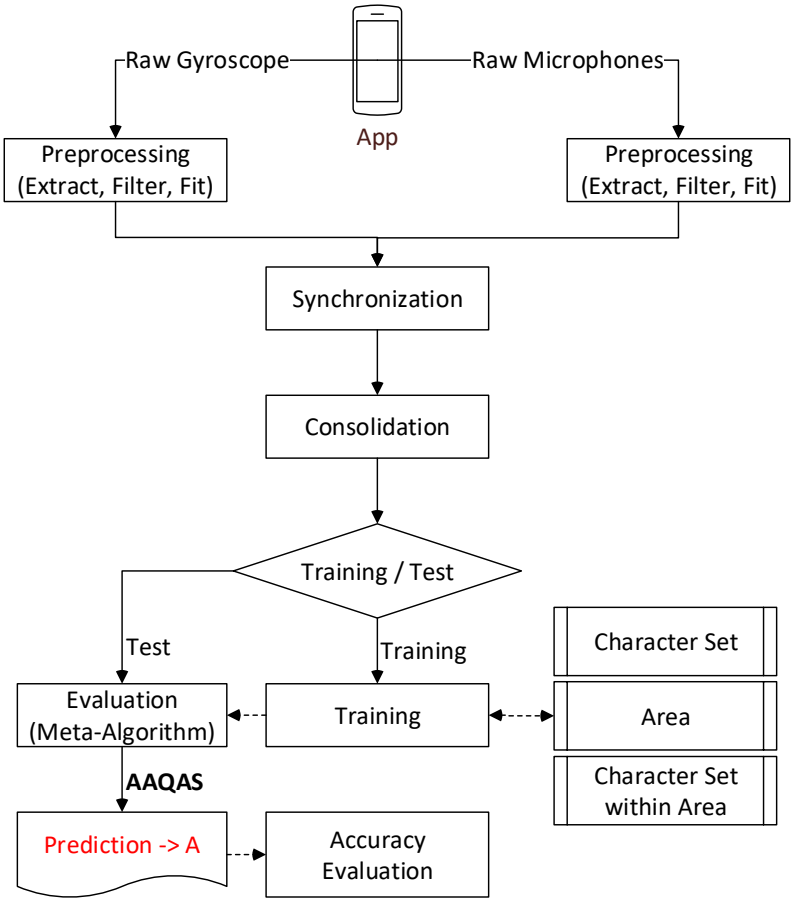


Figure 2.4: Architecture of the keystroke inference system.

server that implements the remaining components of our system to process this data, create training models and perform evaluations on the test data. This raw data contains user keystroke vibrations and sounds mixed with pauses and noise from external factors such as unstable hands (Gyroscope noise) or background music (Audio noise), see Figures 2.6 and 2.7. The *Preprocessing* component removes noise from this data using adequate filtering, extracts the user keystrokes, and performs fitting to resample the data. The techniques used for filtering and extracting Gyroscope and Audio data are different and require two separate preprocessing components. The Gyroscope and Audio data may be combined for inference which also means that the two sets of data need to be time synchronized with each other. The *Synchronization* component synchronizes the Gyroscope and Audio data and then the *Consolidation* component merges the extracted, filtered and

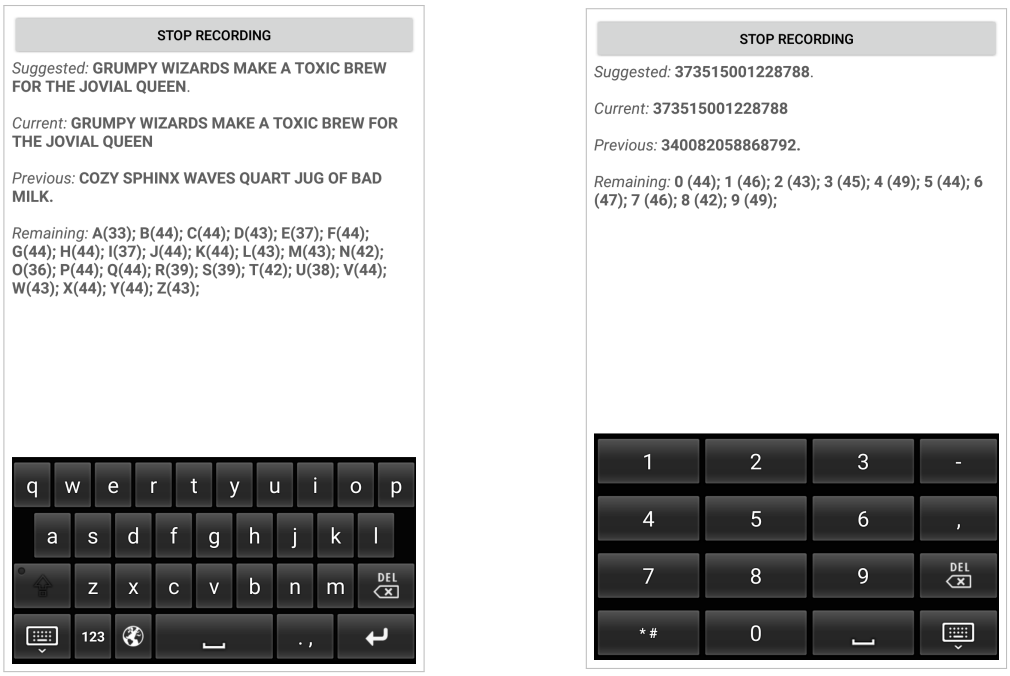
synchronized keystroke data. In order to classify unknown test data, the consolidated data is used to build training models. The *Training* component analyzes the data for errors, randomizes them and uses several machine learning algorithms to build different inference models for the entire character set as well as specific areas on the keyboard. These models are then used by the *Evaluation* component to perform predictions on the unknown test data. The component uses a meta-algorithm that makes a final keystroke prediction. The meta-algorithm uses a multi-step approach based on the specific layout of Android QWERTY and Number keyboards to optimize the inference accuracy of test keystrokes. Once predictions have been made for all unknown test data, the *Accuracy Evaluation* component compares the predictions with the expected keystroke to evaluate our system's performance.

2.3 System Design and Algorithms

In this section, we describe the hardware used and software developed for the keystroke inference system, the data collection process, the Gyroscope and Microphone noise filtering and keystroke extraction process, the training process and the meta-algorithm that was designed and implemented to make predictions on unknown test data.

2.3.1 Data Collection

Hardware & Software: We chose to evaluate our system and meta-algorithm on two popular Android smartphones: a Samsung S2 running Android `Lollipop` and a HTC One running Android `Marshmallow`. We developed an Android app that ran in two separate modes for training and test data, and collected the Accelerometer, Gyroscope and Microphone data in both the modes. The sensor data recorded during the training mode was used for generating the inference models while the data collected in the test mode was used for making predictions from these models. The key difference is that the app invokes a background service in case of the test mode, to simulate a Trojan like behavior when a sensitive app is in the foreground. These two modes are completely independent of each other and do not overlap. This app also implemented a custom keyboard with the same layout and capabilities as the standard Android QWERTY and



(a) App using the QWERTY keyboard

(b) App using the Number keyboard

Figure 2.5: Screenshots of the data collection app using both the standard QWERTY and Number keyboards.

Number keyboard. The key difference here is that our keyboard has the capability to detect a key press and inject this key press event into the Gyroscope recording.

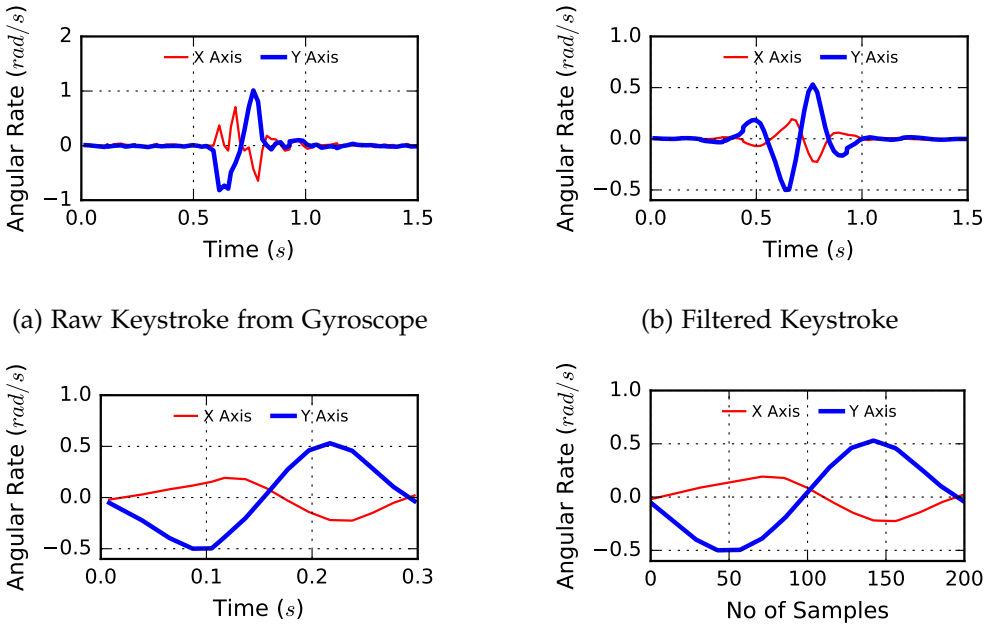
Figure 2.5 shows the screenshot of the app and the custom QWERTY and Number keyboards. The user presses a 'Start Recording' button and types data on the keyboard. The training data is a set of pangrams (a sentence that contains every letter of the alphabet) for the QWERTY keyboard and a set of randomly generated credit card numbers for the Number keyboard. After typing their data, the user presses the 'Stop Recording' button that is visible after recording is started. During this typing session, the Accelerometer, Gyroscope and Microphones are activated in the background and their data are recorded and uploaded to the colluding server. Note that the Accelerometer data was collected only during the initial phase of the experiment. We did not use this sensor subsequently because its accuracy was significantly lower than the Gyroscope. The Accelerometer was extremely sensitive to noise even when the phone was placed stationery on a table.

Sensor Data Collection: The data was collected by seven participants using both the QWERTY and Number keyboards in Portrait mode. About 50 samples per character and number were collected from each participant for training the models. These participants were asked to type in their normal style. We observed their typing behavior and all of them held the device in one hand and typed using the index finger or thumb of the other. One participant held the device in their right hand and the remaining in their left hand. The main difference in typing behavior was the intensity with which the finger touched the screen and the angles at which these devices were held.

The data on the HTC One was collected in two environments. Five participants typed in a typical office environment and two of them also typed in a restaurant. The office environment consisted of a cubicle with three computers and a server running all the time, with additional noise from keyboards, doors opening and closing, people talking and faint noises of vehicles from a nearby street (noise level around 49-52 dB). The restaurant was much more noisier with background music, several people talking and noise from utensils (noise level around 72-76 dB).

The data collected for the Gyroscope sensor were the keystroke's timestamp, the Gyroscope accuracy reported by the operating system, the x, y and z axis orientation values at the maximum sampling rate for the device. We chose to discard the z axis values as vibrations caused by keystrokes mainly affected the x and y planes. The data collected from the Microphones were the *channel* 1 and *channel* 2 amplitudes recorded by the two microphones. A sampling rate of 48 KHz was chosen as it is the maximum sampling rate supported by Android.

Synchronization: The Gyroscope and Audio data are synchronized by injecting a microphone start event in the Gyroscope data. The Trojan app starts the Gyroscope and Microphones in separate threads, however, the microphones are started only once the Gyroscope has completely initialized. This is done to ensure that the Gyroscope sensor starts at the highest sampling rate and does not get reduced to a lower rate due to resource consumption by the microphones. Once the microphone starts recording, the app injects a start event into the Gyroscope data. The two are then recorded simultaneously.



(a) Raw Keystroke from Gyroscope

(b) Filtered Keystroke

(c) Extracted Keystroke

(d) Interpolated Keystroke

Figure 2.6: The stages of preprocessing (filtration, extraction and interpolation) for a Gyroscope recorded keystroke.

2.3.2 Gyroscope Data Preprocessing

Contrary to previous works [6, 7, 10] that extracted features from the Gyroscope recordings, we use the raw Gyroscope x and y axes orientations as the feature set for machine learning. The Android Sensor API returns the rate of change of orientation in *radians/second*. This raw data is filtered, extracted, interpolated and converted to a machine learning compatible format. Figure 2.6 shows the stages of preprocessing for a recorded Gyroscope keystroke, these stages are described below.

Filtration: The Gyroscope noise induced by typing occurs mainly because of the instability of the hand, and this noise is typically high frequency. When this noise is much lower than the change in orientation, i.e., high signal-to-noise ratio, it can be removed by simple frequency filtering techniques. We use a Fast Fourier Transform filter [69] to detect frequencies corresponding to the keystroke. We keep these frequencies unchanged and zero out the amplitudes at the other frequencies. Applying an Inverse Fast Fourier

Transform on this data yields the filtered keystroke data. This technique works quite well, however, advanced filtering schemes such as Kalman Filtering [70] may be applied for noisier data. When the noise is almost the same as the change in orientation, i.e., low signal-to-noise ratio, filtering may remove significant keystroke specific information reducing the inference accuracy. One option would be to use the unfiltered raw data but we observed that the accuracy with filtered data is much better than raw data. This is because the noise in the data changes the waveform and makes them dissimilar even for the same keystroke location. Another option is to observe the Gyroscope vibrations and record only when the noise is under a threshold.

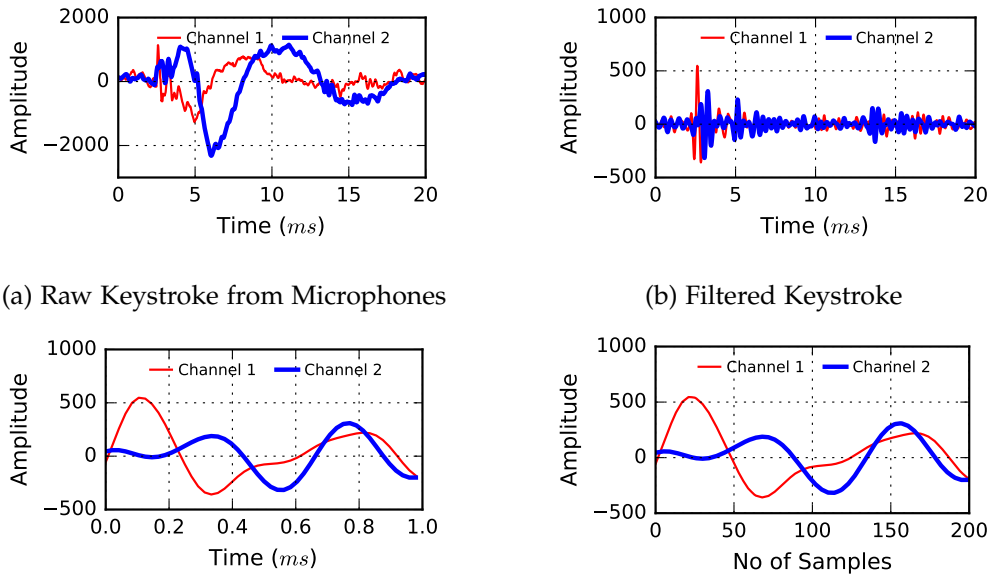
Extraction: The Trojan app we developed implements a custom keyboard for keystroke detection, with the same layout as the standard Android keyboard. This keyboard injects an event into the Gyroscope data when a key is pressed. Our system uses this as the start of the keystroke and a constant time difference to compute the end of the keystroke. For test, the peak amplitude of the audio data was used to detect the start of the keystroke.

Interpolation: The extracted Gyroscope keystrokes are of different sizes because the Android SDK does not allow setting a fixed sampling rate for sensors. As machine learning requires fixed length features, this data has to be resampled before training and making predictions. Another reason for resampling is to increase the size of the data such that minute changes are identified by the algorithms. We use Cubic Spline Interpolation [71] to interpolate the data without changing the waveform of vibrations. Our evaluation results confirm that greater number of samples increase the inference accuracy.

2.3.3 Audio Data Preprocessing

Similar to Gyroscope preprocessing, we use the raw audio recorded by the two microphones (*channel* 1 and *channel* 2) for machine learning instead of extracting features. This raw audio data is filtered, extracted, interpolated and converted to a machine learning compatible format. Figure 2.7 shows the stages of preprocessing for a recorded Audio keystroke, these stages are described below.

Filtration: The noise in audio signals can be due to several external environmental fac-



(a) Raw Keystroke from Microphones

(b) Filtered Keystroke

(c) Extracted Keystroke

(d) Interpolated Keystroke

Figure 2.7: The stages of preprocessing (filtration, extraction and interpolation) for a Microphones recorded keystroke.

tors such as background music, human conversations, and moving traffic. This noise is typically high frequency that can be reduced by using frequency filtering techniques. To extract keystrokes from noisy Audio data, we implemented a bandpass filter to pass all frequencies in the range of 1.5 KHz to 3.5 KHz and filter out the remaining frequencies. This frequency range was obtained by analyzing recordings on the HTC One such that background noise was removed while retaining the frequencies of the tap sound. Bandpass filtering is an effective technique when the keystroke tap and the noise frequencies don't overlap. A low signal-to-noise ratio can have a significant impact on the inference accuracy and even when filters exist to remove noise, the noise removal algorithm may also change the original waveform and decrease inference accuracy. One option is to observe the microphones data and record only when the noise is under a threshold.

Extraction & Interpolation: We use the peak amplitude of the audio data to detect the start of the keystroke. The channel with the higher amplitude becomes the base channel and about two wavelengths are extracted to ensure that the peak of the other channel is

Table 2.1: Accuracy of elementary Machine Learning algorithms for some sample sets.

Keyboard	Sensor	DT	NB	NN	10-NN
		HTC One			
QWERTY	Microphones	86%	85%	90%	80%
QWERTY	Combined	85%	81%	89%	84%
Number	Microphones	70%	81%	72%	66%
Number	Combined	68%	73%	78%	71%
		Samsung S2			
QWERTY	Gyroscope	60%	61%	58%	52%
Number	Gyroscope	74%	74%	82%	72%

also extracted. The extracted keystroke is then interpolated so that minute changes in the data are detected as features by the machine learning algorithms.

2.3.4 Training Process

The keystroke inference system uses the specifics of Android QWERTY and Number keyboard, and a number of steps and algorithms to develop adequate training models.

Consolidation: The filtered, extracted and interpolated Gyroscope and Microphone keystrokes are consolidated to represent the keystroke's features for machine learning. Given N samples, the Gyroscope keystroke is represented as the list of all x and y orientations, i.e., $[x_0, x_1, \ldots, x_{N-2}, x_{N-1}, y_0, y_1, \ldots, y_{N-2}, y_{N-1}]$. Given N samples, the Audio keystroke is also represented as the list of all channel 1 ($c1$) and channel 2 ($c2$) amplitudes, i.e.,. $[c1_0, c1_1, \ldots, c1_{N-2}, c1_{N-1}, c2_0, c2_1, \ldots, c2_{N-2}, c2_{N-1}]$. Note that N is the same for both Gyroscope and Audio keystrokes, to give both these sensors an equal weight during inference. These two lists of features are simply merged together when *Combined* (Gyroscope + Microphones) data is desired for devices supporting stereo-Microphones.

Elementary Algorithms: The problem of inferring unknown user keystrokes using models generated from a training set of known user keystrokes is a supervised classification problem. Therefore, we eliminated algorithms that are used in unsupervised classification (e.g., K-Means). The algorithms we tested were Decision Trees, Naive Bayes, K-Nearest Neighbor (k-NN), Hidden Markov Models, Support Vector Machines, Random

(a) Area division of the QWERTY keyboard (b) Area division of the Number keyboard

Figure 2.8: Screenshots of the area divisions used by the Meta-Algorithm for both the QWERTY and Number keyboards.

Forest and Neural Networks. From the above list, *Decision Trees (DT)*, *Naive Bayes (NB)*, *1-Nearest Neighbor (NN)* and *10-Nearest Neighbor (10-NN)* performed better and yielded higher inference accuracies. Neural Networks also performed well but was discarded due to heavy resource consumption. In the context of our work, instance-based methods such as k-NN yield high accuracy as they try to find the closest match between the new prediction and the training data. Table 2.1 shows the performance of these elementary algorithms for three QWERTY and three Number keyboard sample sets. The results show that none of these algorithms perform well on all areas of the keyboard because of overlapping instances. This observation drove us to design and develop a Meta-Algorithm which considers the areas of keyboard before making predictions.

Area Division: The keyboards were divided into areas such that all keys in a specific area can be distinguished from each other using at least one set of features. The area division for a standard QWERTY keyboard in portrait mode is shown in Figure 2.8a. These areas are chosen such that the y orientation differs for all keys in the area, while the x orientation remains similar. For instance, the negative y orientation will be higher for 'Q', lesser for 'W' and the least for 'E' in the area 'QWE'. The area division for a standard Number keyboard in portrait mode is shown in Figure 2.8b and follows the same reasoning except for keys '8' and '0', where the x orientation will differ while y orientation remains similar. We tested other area divisions as well but this division worked better than the other divisions.

The Training Process: The goal of the training process is to build inference models for

the entire character set as well as areas defined for the keyboard. The following steps describe this process.

I. Training models are built for predicting areas of the keyboard using a voting [72] algorithm. This algorithm uses the predictions of all provided algorithms and their confidence values to determine the area of the test keystroke. The voting model uses ensembling techniques such as Bagging and Boosting to improve the accuracy of the predictions. Ensembling builds multiple models from subsets of the training data, analyzes the accuracy of the subsets to detect incorrectly classified instances, and then uses these instances again with different weights or averaging to build better predictive models.

II. Training models are built for the entire character set using all the algorithms. These models also use ensembling. If the training data is *Combined* (Gyroscope + Microphones), then training models for Microphone data are also built. This is because the Gyroscope data for certain areas of the keyboard that are close to the actual hardware may be weaker that other areas, and this weak Gyroscope data may reduce the overall accuracy of the model for that area.

III. Training models are built for all character sets within an area, for each area. These models also use ensembling. If the training data is *Combined* (Gyroscope + Microphones), then training models for Microphone data are also built. Our system evaluates these models using multi-fold cross validation. In cross validation, a subset of the training data is provided to the model as test data and the accuracy of the model is computed. By using multiple folds, a model can be tested multiple times with different training data to ensure generalization and their accuracies are averaged. The system uses this accuracy to determine which models are better for an area and uses these models for predictions for that area.

IV. The two best algorithms for an area (determined in step III) are used to form a voting algorithm. This voting algorithm is used to make the final prediction in case all previous steps do not converge on a prediction. In case the algorithms predict different keystrokes, their confidence values are used to make the final prediction.

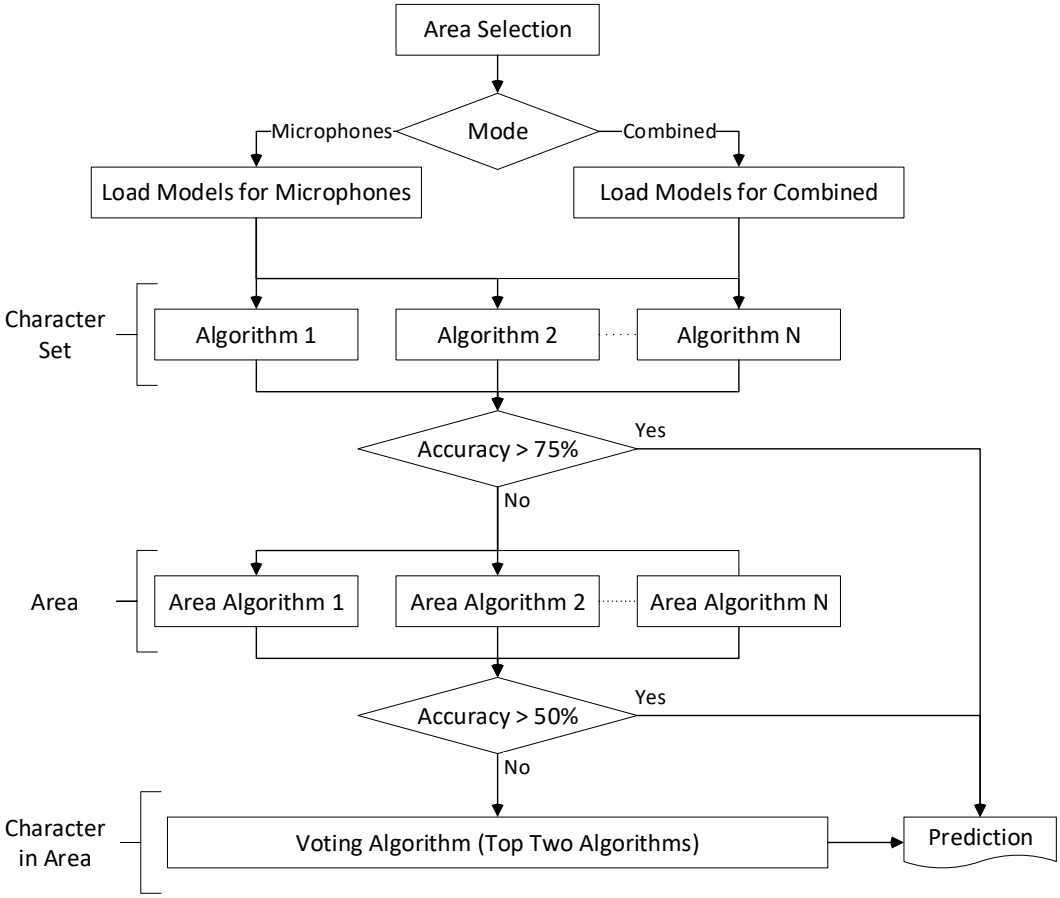


Figure 2.9: Flow diagram of the Area-Based Meta-Algorithm for keystroke inference.

2.3.5 Area-Based Meta-Algorithm

The keystroke inference accuracy achieved using elementary algorithms was not high (cf. Table 2.1), and different algorithms predicted different keys for the same test keystroke. To address this, we developed a Meta-Algorithm that utilizes our area specific models at multiple levels to infer keystrokes with higher accuracy than traditional algorithms. Our evaluation demonstrates that the Meta-Algorithm yields much higher accuracy. Figure 2.9 shows the flow diagram of our Meta-Algorithm and the levels of evaluations. The following steps describe the process of keystroke inference using this algorithm.

I. The test keystroke is first evaluated using the area voting model. The goal of this step is to identify the area of the keystroke and load the appropriate models for that

area. The system evaluates the models built for every area and maintains a list of the models that have yielded high accuracy with the training data for that area. For instance, the area 'IOP' on the QWERTY keyboard of a HTC One may have weak Gyroscope data (as it is close to the Gyroscope hardware) implying that the *Combined* model will also be weak. The system detects this and loads the Microphone only model for evaluation instead of the combined model.

II. The test keystroke is then evaluated using the loaded character set models. We use these models before the area-specific models to ensure that any prediction errors by the voting model (in step I) is detected and corrected. For example, the voting model may have predicted the area of a test keystroke as 'IOP' when the actual key pressed was 'K'. This is possible because of the presence of weak or noisy Gyroscope data in the voting model. The system will load the Microphone models when the Gyroscope data is weak and can then evaluate the test keystroke correctly based on the audio data. If more than 75% of the models predict the same key, then this key is chosen as the final prediction and no additional steps are performed.

III. The test keystroke is then evaluated using the area-specific models. This step is only executed when the character models (in step II) were not successful in predicting the keystroke. One reason could be the prediction of neighboring keys which belong to another area. For example, a character set based model may predict the key as 'A' when the actual key is 'Q'. These keys are neighbors on a standard Android QWERTY keyboard and may contain similar vibrations and audio characteristics. When the test data is evaluated specifically using the models for area 'QWE', then they can only predict a keystroke from this area and may predict the correct keystroke. If more than 50% of the models predict the same key, then this key is chosen as the final prediction and no additional steps are performed.

IV. The test keystroke is finally evaluated using a voting model consisting of the two best algorithms for that area. This model outputs the final prediction based on the prediction and confidence values of the two loaded algorithms. This step will generally be executed when the test data is quite noisy and difficult to infer. We do

Table 2.2: Area-wise accuracy on a QWERTY keyboard for a HTC One sample set.

Area	Gyroscope	Microphones	Combined
Q, W, E	84%	90%	92%
R, T, Y, U	86%	86%	92%
I, O, P	79%	90%	99%
A, S, D	84%	94%	97%
F, G, H	70%	89%	92%
J, K, L	71%	84%	89%
X, Z	88%	80%	98%
C, V, B	83%	93%	93%
N, M	77%	90%	100%

not discard the data but attempt to make a final prediction based on the two best algorithms for that area.

2.4 Evaluation

We evaluate the keystroke inference system using the following metrics: the performance of the Gyroscope, Microphones and Combined (Gyroscope + Microphones) sensors for different areas of the keyboard; the performance of the Meta-Algorithm for individual Machine Learning algorithms in comparison to traditional use of these algorithms; and the performance of the Meta-Algorithm for all the collected user sample sets.

2.4.1 Impact of Sensors on Keyboard Areas

Table 2.2 shows the area-wise inference accuracy of the Gyroscope, Microphones and Combined sensors for a sample set collected on the QWERTY keyboard in portrait mode, on the HTC One. The table shows that the Gyroscope predictions are inconsistent across areas as compared to the Microphones which is consistent throughout. This is because the Gyroscope vibrations are dependent on the location of the key with respect to the Gyroscope hardware. The areas 'IOP', 'ASD' and 'NM' are close to the Gyroscope hardware and do not exhibit significant vibration on the y axis. Their inference depends more on the x axis vibrations yielding lower accuracy for these areas. The areas 'XZ', 'ASD', and 'QWE' are further from the Gyroscope hardware and exhibit significant vibrations on the

Table 2.3: Accuracy of the Meta-Algorithm when applied to individual Machine Learning algorithms for some sample sets.

Keyboard	Sensor	DT	NB	NN	10-NN
		HTC One			
QWERTY	Microphones	94%	86%	93%	85%
QWERTY	Combined	95%	80%	93%	91%
Number	Microphones	80%	81%	79%	76%
Number	Combined	81%	77%	81%	77%
		Samsung S2			
QWERTY	Gyroscope	68%	61%	60%	55%
Number	Gyroscope	82%	76%	84%	79%

y axis. Their inference depends on vibrations in both x and y axes yielding higher accuracy for these areas. Microphone predictions, on the other hand, are location independent as they rely on the speed of sound traveling over the surface. The table also shows that the accuracy of the microphones is higher than the Gyroscope in most of the areas, the only exception being area 'XZ'. Combining the data from the two sensors yields higher accuracy than the individual sensors in cases when the Gyroscope data for an area is not weak. In situations where the Gyroscope data is weak, our system attempts to detect this and performs inference using just the Microphone models for that area.

2.4.2 Impact of Meta-Algorithm on Individual Algorithms

Table 2.3 shows the performance of the Meta-Algorithm when applied to individual Machine Learning algorithms, compared to the elementary use of these algorithms (cf. Table 2.1). Note that the same sample sets are used in both the evaluations so that the results can be directly compared. The table shows that our Meta-Algorithm improves the inference accuracy for every sample set. This improvement varies for every Machine Learning algorithm. The table shows that the Decision Tree (DT) algorithm benefits the most from the Meta-Algorithm, with high increase in inference accuracies (8% - 13%). The Naive Bayes (NB) algorithm does not benefit much from the Meta-Algorithm, with low increase in inference accuracies (0% - 4%).

Table 2.4: Final Single-stroke Meta-Algorithm accuracy for all sample sets collected in the Office environment.

User	Keyboard	Count	Gyroscope	Microphones	Combined
		HTC One			
User1	Number	306	68%	93%	93%
User2	Number	200	44%	94.5%	93%
User3	Number	300	72%	91%	91%
User4	Number	300	75%	94%	95.5%
User5	Number	323	45%	83%	83%
User3	QWERTY	782	80.5%	89.5%	94%
User4	QWERTY	860	56%	83%	83%
User5	QWERTY	877	66%	73.5%	84%
		Samsung S2			
User1	Number	137	75.5%	-	-
User2	Number	542	84%	-	-
User3	Number	202	83%	-	-
User4	Number	200	81.5%	-	-
User5	Number	512	81%	-	-
User1	QWERTY	366	63.5%	-	-
User2	QWERTY	620	77%	-	-
User5	QWERTY	312	74%	-	-

2.4.3 Performance of the Meta-Algorithm

Table 2.4 shows the final inference accuracy of the Meta-Algorithm for all sample sets collected in the Office environment. The table shows that it is possible to achieve high accuracy using just the stereo-Microphones of the device. We achieved an inference accuracy of 89.5% on the QWERTY keyboard for User3, and an accuracy of 94.5% on the Number keyboard for User2. In some cases such as the QWERTY keyboard sample for User3, combining the Gyroscope and the Audio data can boost inference and it is possible to reach an accuracy of 94% even on the QWERTY keyboard. In some cases such as the Number keyboard for User2, combining the data may also result in a decrease in accuracy. This is likely when the Gyroscope data is weak, does not contain significant vibrations or contains significant noise. We built our system to detect such weak Gyroscope data

using cross-validation of training samples but we did come across situations when the cross-validation yielded high accuracy for weak Gyroscope data. One alternative could be to manually define which models should be loaded for specific areas using the knowledge of the Gyroscope hardware location for that smartphone model. There were some sample sets where the Gyroscope inference accuracy was as low as $44 - 56\%$. We evaluated them manually to find that our filtering techniques were unable to handle large Gyroscope drifts. These drifts can be compensated by using a Kalman filter combining the Accelerometer and Gyroscope data.

2.4.4 Impact of Noise on Meta-Algorithm

We also evaluated our keystroke inference system in scenarios when the attack may not perform so well. An example of external environmental scenarios affecting inference are very noisy restaurants, while an example of non-environmental scenarios is soft typing. To evaluate these scenarios, we asked two participants to collect keystrokes in a restaurant with noise levels between 72 dB and 76 dB. Using just the Microphones, the system achieved an accuracy of 42% and 56% for 212 and 226 test keystrokes, respectively. Another two participants were chosen who typed with very soft touches. In their case, the system achieved a low accuracy of less than 20% using both the Gyroscope and Microphones data. A manual analysis of the data revealed that the keystrokes could not be differentiated from the background noise. We also asked two participants to collect keystrokes on a Samsung Tab 3 tablet, and achieved an accuracy of 36% and 45% for 106 and 234 test keystrokes, respectively. Note that this inference was made using just Gyroscope data as the tablet did not support stereo recording. Both participants held the tablet in two hands and typed with their thumbs, reducing the vibrations caused by typing.

2.4.5 End-to-end Attack Evaluation

To illustrate an end-to-end attack, we modified our Trojan app to query the foreground activity every five seconds. In Android, every UI page is known as an activity. An app may have multiple activities, each with a different class name. An adversary can easily determine the functionality of an app using these class names. For example, the login activity

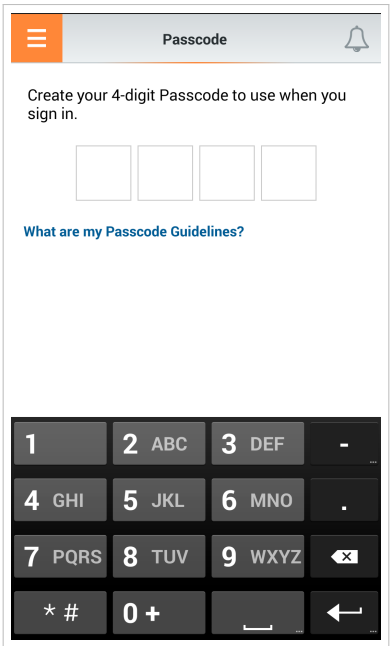


Figure 2.10: Screenshot of the bank activity used for demonstrating the end-to-end attack.

Table 2.5: Final Meta-Algorithm accuracy for 100 random PIN numbers and 100 random Credit Card numbers.

Total	**Correct**	**Correct Digits**	**Accuracy**
		PINs	
100	84	376	94%
		Credit Cards	
100	52	1467	91.5%

is named as com.*****.mobile.LoginActivity for the banking app used for demonstration (parts of the class name hidden here for anonymity of the app). Figure 2.10 shows the screenshot of this activity. The Trojan starts recording the Gyroscope and Microphones when it detects a login activity or credit card input activity in the foreground.

We collected 100 four digit random numbers and 100 sixteen digit random numbers simulating PINs and credit card numbers from our participants. Table 2.5 shows the accuracy reported by the Meta-Algorithm for these PINs and credit card numbers. For four digit PIN numbers, the system correctly predicted 376 digits out of 400 digits and 8

additional keystrokes were detected by the system. Out of the 100 PIN numbers, 84 were predicted completely correctly in a single attempt. For sixteen digit credit card numbers, the system correctly predicted 1467 digits out of 1600 digits and 12 additional keystrokes were detected by the system. Out of the 100 credit card numbers, 52 were predicted completely correctly in a single attempt.

2.4.6 Impact of Assumptions on Attack Performance

In this subsection, we discuss the impact that the assumptions made for this work may have on the attack performance.

Training Data Collection: In this work, we assume that the Trojan app collects at least 50 keystrokes for each character from the user. These keystrokes are then used as the training data to build generalized training models. This assumption can be easily realized by an adversary by implementing and distributing a note-taking app, or simply a typing tutor app that customizes the sentences to build the training data fairly quickly.

Known Keyboard Layout: Our assumption of known keyboard layout does not impact attack performance because the training models and the attack use the same device and, therefore, the same keyboard layout. This simplifies the attack as the adversary does not have to worry about customization for different devices having different form-factors and hardware layouts. Even when the operating system is updated to another version, the key sizes of the Android keyboard remain standard to ensure usability and preventing the users from re-learning a new layout.

2.5 Related Work

Cai & Chen [6] were the first to study the feasibility of number keystroke inference attacks using an Android device's orientation sensor. They developed an Android app called TouchLogger and collected three data-sets on a HTC Evo 4G phone using a Number only keypad in Landscape mode. Their experiments achieved a successful inference accuracy of 71.5% for all three data-sets and demonstrated that such an attack was feasible.

Owusu et al. [9] studied the feasibility of area and character inference using an An-

droid device's Accelerometer sensor. They developed an Android app called ACCessory for collecting data-sets on a HTC ADR 6300 phone from four participants. The participants were instructed to hold the phone and enter keys in a certain manner and several data-sets were collected for screen area and characters using a QWERTY keypad in Landscape mode. The data-sets were used to build a predictive model to evaluate the accuracy of area and passwords inference. Their experiments showed that, out of 99 6-character passwords, it was possible to successfully infer 6 passwords in 5 trials.

Xu, Bai & Zhu [7] used two motion sensors, Accelerometer and Orientation, to study the feasibility of inference of the lock screen password and the numbers entered during a phone call, such as credit card and PIN numbers. They developed an Android app called TapLogger that stealthily logs these numbers by using the Accelerometer sensor to detect the occurrence of taps and the Orientation sensor to infer which number was typed by the user. They collected data-sets of several tap events from three students using two phones, HTC Aria and Google Nexus (One), and unlike other experiments, performed the training and classification on the smartphone itself. Their experiments achieved an accuracy of about 99% for one user on the Google Nexus (One) and about 70% - 85% accuracy for the other users.

Cai & Chen [73] studied the impact of different settings on the accuracy of keystroke predictions. They varied different factors in their settings, such as user habits, screen size, device type, layout orientation, etc. Their results show that side channel attacks stay possible and practical regardless of the setting. Although the attacks are feasible, the inference accuracy varies. They used Google Nexus S, HTC Evo 4G, Galaxy Tab 10.1, Motorola Xoom in their experiments with 21 users, and demonstrated that 4 digit PIN can be guessed correctly after 81 attempts, 65% of times.

Aviv et al. [8] demonstrated the possibility of inferring PIN and pattern password by exploiting the Accelerometer on four different smartphones; Nexus One, G2, Nexus S and Droid Incredible. Their results and evaluations were based on 24 users, divided into two groups of 12. Each group performed controlled (seating) and uncontrolled (walking) experiments. In the controlled setting, they reached an accuracy of 43% and 73% for

PIN and pattern passwords respectively, within 5 attempts from a set of 50 PINs and 50 patterns. In the uncontrolled setting, they could predict PINs and patterns 20% and 40% of times respectively, within 5 attempts.

Miluzzo et al. [10] presented a framework called TapPrints that used the Accelerometer and Gyroscope to identify icon locations and infer characters typed on a keyboard. They collected a data-set on two Android devices, the Google Nexus S and Samsung Galaxy Tab 10.1, and a iPhone 4. The experiment with icon locations was performed with the device in Portrait mode while other experiments with the character keypad were performed with the device in Landscape mode. By using ensemble machine learning, they showed that on an average, locations of icons can be inferred with 79% and 65% accuracy for the iPhone and Google Nexus S respectively and characters could be inferred with 65% accuracy. They also showed that some icons or characters can be inferred with an accuracy of up to 90% and 80% respectively.

Marquardt et al. [13] demonstrated that an Android app with access to the device's Accelerometer can be used to recover text typed on a physical keyboard the device is placed in close proximity with. They showed that if a device is placed within 2 inches of a physical keyboard and the keyboard is used for typing, then by measuring the relative physical position and distance between the vibrations, they could recover words with accuracy as high as 80%.

Templeman et al. [11] proposed a proof-of-concept visual malware called PlaceRaider. It opportunistically used the camera and other sensory data from a smartphone to create a 3D model of the user's environment. This 3D model allows the adversary to navigate and zoom in areas of interest to examine the individual images corresponding to that region. Another example of sensory malware is Soundcomber [74] which uses microphone to steal private information such as credit card numbers from phone conversations.

Table 2.6 summarizes the inference accuracy and the attack characteristics (i.e., the smallest keyboard used for the attack, the key size ratio compared with QWERTY keyboard in portrait mode, and number of attempts required) of the closely related works in comparison to our attack. We must note the following regarding key size ratio: (1)

Table 2.6: Summary of the attack characteristics and inference accuracy of related works in comparison to our keylogging attack.

	Smallest Keyboard	Size Ratio	Attempts	Inference Accuracy
Aviv et al. [8]	Lock Screen	> 4.0	5	43% of 50 4-digit PINs
TapLogger [7]	Lock Screen	> 4.0	1	99%, 75%, 84% for 4-digit PINs for 3 users
TouchLogger [6]	Number in Landscape	~ 2.9	1	71.5% for individual digits
Cai & Chen [73]	Number in Portrait	~ 2.3	81	65% for 4-digit PINs
ACCessory [9]	QWERTY in Landscape	~ 1.2	5	6 of 99 6-character passwords
TapPrints [10]	QWERTY in Landscape	~ 1.2	1	65% for individual characters
This Attack	QWERTY in Portrait	1	1	83%-95% for individual characters

this ratio is an approximation of the area of the keys on the current Android keyboards, and (2) the key sizes used in the related works are for older keyboards that were larger than the current keyboards. Our keystroke inference system performs significantly better even with smaller key sizes than other works. It also differs from previous works as it uses the stereo-recording capability of smartphones, a combination of sensor and acoustic information, adequate sensor and audio noise filtering, keyboard specific information, domain-specific machine learning, and a multi-tier approach in the Meta-Algorithm. This combination achieves a higher inference accuracy for the standard Android keyboards.

2.6 Conclusion

In this chapter, we investigated the feasibility of keystroke inference when user taps on a soft keyboard of a smartphone are captured by the *Gyroscope* and *stereoscopic Microphones* sensors co-resident on the smartphone. We demonstrated that it is possible to infer keystrokes with an accuracy of 90-94% on the standard Android QWERTY and Numeric keyboards by using a combination of multiple sensors, adequate filtering, and building machine learning models specific to the keyboard.

Chapter 3

Inferring User Routes and Locations

This chapter describes our location tracking attack that exploits the zero-permission Accelerometer, Gyroscope and Magnetometer sensors, and public map resources to track vehicular users. We developed a framework to assess the feasibility of this attack and optimize tracking by (1) creating architecture blocks that addresses challenges in inference, (2) modeling location tracking from sensors as a graph theoretic problem, and (3) developing efficient search algorithms that maximize probability of finding the traveled route in a small set of results. In Section 3.1, we describe a scenario of how a stealthy attack can be launched by a victim inadvertently downloading a malicious app. In Section 3.2, we formalize the problem, identify the challenges, define the adversarial model, and describe the high-level architecture of our system. Section 3.3 provides a detailed description of our graph construction technique, and design of the search algorithms and filters. In Section 3.4, we outline the evaluation metrics and present the results of our evaluations. In Section 3.5, we describe previous related work and we conclude in Section 3.6.

3.1 Attack Vector

The victim is engaged in the act of driving a vehicle where they are co-located with an active smartphone. The adversary's goal is to track the victim without the use of traditional position determining services such as GPS, cell tower pings, or Wi-Fi / Bluetooth address harvesting. To prepare for an attack, the adversary uploads a seemingly innocuous mobile app to a publicly accessible Application Store. The app is subsequently downloaded and installed by the victim on their smartphone. While providing the victim with its advertised features, this malicious app additionally collects sensor data from the Accelerometer, Gyroscope and Magnetometer.

The attack is triggered when the app detects that a victim is starting to drive. Sensor

data is recorded in the background, without visible indication of the recording activity, and uploaded to a colluding server whenever Internet access is available. Based on this sensor data, the adversary can derive driving information such as turn angles, route curvatures, accelerations, headings and timestamps. Combined with publicly available geographic area attributes, the adversary can learn the actual route taken without the need of any location services / information.

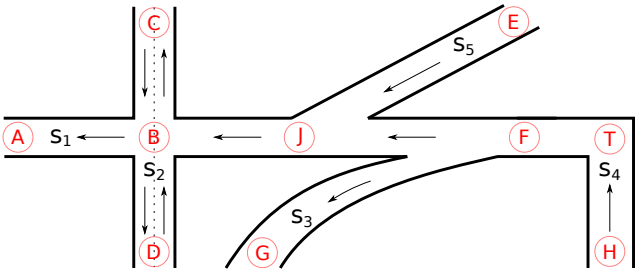
3.2 High-Level Approach

In this section, we formalize the problem of location tracking using zero-permission sensors, outline some of the challenges in tracking, define the adversarial model and describe the architecture of our location tracking system.

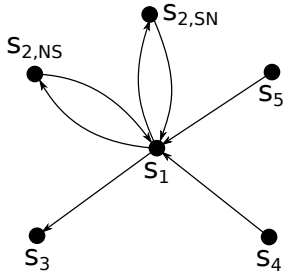
3.2.1 Location Privacy Leakage from Sensor Data

We introduce our terminology and notations used to describe the problem space. Consider a geographic area represented by a set of roads. Each road is either straight or has curvature that is detectable by the smartphone's sensors. A connection is created when a road bisects, furcates, joins with other roads, or turns into a different direction (cf. Figure 3.1a). These connections divide roads into multiple *atomic parts*, which only connect with other atomic parts at their end points. Therefore, a geographic area $\mathcal{G}$ can be uniquely described as $\mathcal{G} = (\mathcal{B}, \mathcal{C}, \theta, \vartheta)$, where $\mathcal{B}$ is a set of atomic parts, and $\mathcal{C} = \{\chi = (r, r') | r, r' \in \mathcal{B}\}$ consists of ordered pair of connections $\chi = (r, r')$ indicating the connection between two atomic parts r and r'. The turn angle associated with a connection χ, which captures the real-world travel direction from r to r', is given by the function θ. A positive angle $\theta(\chi) > 0$ indicates a left turn, and a negative value $\theta(\chi) < 0$ indicates a right turn. Finally, the atomic parts preserve the road curvature determined by $\vartheta(r)$. The computation of θ and ϑ functions is based on the public map information.

We define a route taken by the driver as a sequence $\mathcal{R}$ of connected atomic parts, $\mathcal{R} = (r_1, \ldots, r_N)$, where $(r_i, r_{i+1}) \in \mathcal{C}$. Two routes $\mathcal{R}$ and $\hat{\mathcal{R}}$ are identical if the sequences of atomic parts have the same size and are component-wise equal, i.e., $\mathcal{R} = \hat{\mathcal{R}}$ if $r_i = \hat{r}_i$ for all i. Along the driving trajectory, the app obtains a set of sensor data $\mathcal{D} = \{(a_t, g_t, m_t)\}$



(a) Connections are created when a road bisects (B), furcates (F), joins (J) with another road, or turns (T) into a different direction. Created atomic parts: $\overrightarrow{BA}, \overrightarrow{BC}, \overrightarrow{BD}, \overrightarrow{CB}, \overrightarrow{DB}, \overrightarrow{JG}, \overrightarrow{EJ}, \overrightarrow{FJ}, \overrightarrow{FG}, \overrightarrow{TF}, \overrightarrow{HT}$.



(b) Graph construction: every one-way road segment s_1, s_3, s_4, s_5 is represented by one vertex, while two vertices $s_{2,\text{NS}}$ and $s_{2,\text{SN}}$ are created for the north-south (NS) and south-north (SN) directions of the road segment s_2, respectively.

Figure 3.1: Example of a hypothetical road network, and its mapping to a graph for location tracking.

consisting of the vectors a_t, g_t and m_t taken from the Accelerometer, Gyroscope and Magnetometer respectively. These vectors are sampled according to discrete time periods $t = 0, \delta, 2\delta, \ldots$, where δ is the sampling period. Based on $\mathcal{D}$, an adversary launches the tracking attack as follows.

Definition 1 (Sensor-based Tracking Attack)**.** *Let $\mathcal{A}$ be the attack deployed by the adversary on the received sensor data $\mathcal{D}$ given geographical area $\mathcal{G}$. The outcome of the attack is a ranked list $\mathcal{P}$ of K possible victim routes $\mathcal{P} = \mathcal{A}(\mathcal{G}, \mathcal{D}) = \{\hat{\mathcal{R}}_1, \ldots, \hat{\mathcal{R}}_K\}$, where $\hat{\mathcal{R}}_i$ has higher probability than $\hat{\mathcal{R}}_j$ of matching with the victim's actual trajectory, if $i < j$.*

A successful tracking attack will be one in which a small set of results yield a route list containing the truth route. We aim to design an attack that satisfies this objective with

success probability significantly higher than a random guess. In particular, we evaluate the attack efficiency according to the following metrics.

Definition 2 (Individual Rank)**.** *Given the user's actual trajectory $\mathcal{R}$ and the outcome of the attack $\mathcal{P} = \mathcal{A}(\mathcal{G}, \mathcal{D})$, the individual rank of the attack is k, if $\mathcal{R} = \hat{\mathcal{R}}_k$. The rank is uninteresting if $\mathcal{R}$ is not found in $\mathcal{P}$.*

The individual rank k reflects the attack's success in estimating that the victim's route is in top k of the outcome list. We are interested in the probability of such event happening, i.e., $P_k^{idv} := P(\mathcal{R} \in \{\hat{\mathcal{R}}_1, \ldots, \hat{\mathcal{R}}_k\})$, and evaluate the attack performance based on it (see Section 3.4). While P_k^{idv} shows the possibilities of the victim's route being in a top k rather than telling which among the top is the actual route, the probability P_1^{idv} is precisely the probability of finding the victim's route. This probability, though small (e.g., $P_1^{idv} \approx 13\%$ for Boston and $\approx 38\%$ for Waltham in our preliminary real-driving experiments), is still considerably high given the fact that the search space contains billions of routes. In practice, a top k with small k (e.g., $k \leq 5$) is a very serious breach. An adversary may collect such lists through the span of multiple days and refine the lists to find exactly the victim's daily commute route. Moreover, with more resources, the adversary can quickly check every potential route in the list to learn about the victim.

While the individual rank reflects the performance of the attack in terms of finding the exact route, in practice a rough estimation of the victim's route is usually enough to create a significant privacy threat. For example, targeted criminal activity (i.e., robbery and kidnapping) could result from the physical proximity knowledge derived from the attack. To justify this threat, we define a *cluster* of routes as a set $\{\hat{\mathcal{R}}_1, \ldots, \hat{\mathcal{R}}_l\}$ in which any two routes are similar. The similarity of routes $\hat{\mathcal{R}}$ and $\hat{\mathcal{R}}'$ is defined as $d(\hat{\mathcal{R}}, \hat{\mathcal{R}}') < \Delta$, where $d(\hat{\mathcal{R}}, \hat{\mathcal{R}}') = \sum_{i=1}^{N-1} \| \mathrm{Loc}(\hat{\chi}_i) - \mathrm{Loc}(\hat{\chi}'_i) \|$ is the sum of distances between connection points $\hat{\chi}_i = (\hat{r}_i, \hat{r}_{i+1})$, $\hat{\chi}'_i = (\hat{r}'_i, \hat{r}'_{i+1})$ on $\hat{\mathcal{R}}$ and $\hat{\mathcal{R}}'$, $\mathrm{Loc}(\cdot)$ denotes the geographic coordinates, and Δ is a threshold value.

By clustering, the attack now returns the outcome as another ranked list similar to one in Definition 1. Routes belonging to the same cluster are removed and only the best one of the corresponding cluster is included in the list. Specifically, if $\mathcal{A}_{cluster}(\mathcal{G}, \mathcal{D}) =$

$\{\hat{\mathcal{R}}_1, \ldots, \hat{\mathcal{R}}_K\}$, then $d(\hat{\mathcal{R}}_i, \hat{\mathcal{R}}_j) \geq \Delta$ for any i, j, and $\hat{\mathcal{R}}_i$ is a representative route of cluster i. We now introduce the *cluster rank* metric as follows.

Definition 3 (Cluster Rank). *Given the user's actual trajectory $\mathcal{R}$ and the outcome of the attack $\mathcal{P} = \mathcal{A}_{cluster}(\mathcal{G}, \mathcal{D})$, the cluster rank of the attack is k, if $d(\mathcal{R}, \hat{\mathcal{R}}_k) < \Delta$. The rank is uninteresting if no such k is found.*

Similar to individual rank, we are interested in the probability of a route being in the top k of clusters, i.e., $P_k^{clt} := P(\mathcal{R} \in cluster_1 \cup \ldots \cup cluster_k)$. Based on the cluster rank metric, the adversary may eliminate similar routes and focus computation power on additional routes to improve the search results. Clustering is useful when similar roads / turns are present to effect a nearly identical result. For instance, the adversary may group routes with the same end points while ignoring different roads in between, or if they differ only at one end point (start or end), e.g., roads going from / to residential complex or office areas. This may give the adversary more confidence in an area than the individual rank.

3.2.2 Challenges in Inference

There are several challenges to the attack feasibility including the geographic area size, impact of sensor noise, driver behavior, and road similarity.

Area Size: The geographic area's size has an impact on the attack's accuracy. Even in small cities like Waltham (Massachusetts, USA), there are billions of possible routes for a victim. Moreover, routes with loops may also significantly increase the search space.

Noisy Sensor Data: The quality of sensor data is key for high attack accuracy. Unfortunately, today's smartphones are equipped with low-cost sensors that do not guarantee high accuracy. Sensor accuracy is also dependent on the sensor's previous state, e.g., the acceleration can spike due to a street bump but requires settling time before providing new useful information. The Magnetometer is influenced by nearby magnetic fields from fans, speakers and other electromagnetic devices.

Driver Behavior: The driving style of a driver also impacts the estimation of the actual route. For instance, a driver may frequently speed up or slow down due to traffic conditions or change lanes to overtake other vehicles. These actions induce additional noise in

the sensor data in the form of spatial perturbations or distortions.

Road Similarity: Even in ideal scenarios when clean sensor data is obtained, the similarity of roads impacts the estimation of the actual route. This is especially true for cities with grid-like road structures such as Manhattan, New York.

3.2.3 Adversarial Model

Mobile Application: We assume that the malicious app collects sensor data continuously, either actively or in background, and intermittently transfers the data to the colluding server. As a typical one hour trip collects approximately 800 KB of uncompressed data (80 KB/hour for processed and compressed data), detection by a user in the form of degraded network behavior should be negligible in locations with active 3G and 4G networks or nominal Wi-Fi signal strength.

Device Position: We compensate for device orientation at attack initiation (i.e., the time when the vehicle starts moving). During travel, the device's orientation should remain relatively fixed within the reference frame of the vehicle. This supports attack efficacy in a variety of realistic phone placements such as the phone attached to a mount, residing in a cup holder, in the driver's pocket or in her handbag.

Location Information: While the attack described in this work does not rely on the location information of the victim's trajectory at any point (e.g., no known starting point), we assume a rough knowledge of their living / travel area (e.g., known to live in / frequent Manhattan, New York).

3.2.4 System Architecture

In the most basic form, the system consists of a smartphone that collects data and a post-processing server that generates a ranked list of potential routes or clusters of routes. Figure 3.2 illustrates the design's main components.

- *Preparation:* Road information from public map resources are extracted and converted to specific database structures. This is a one-time initialization step and the structures can be reused for all subsequent attacks.

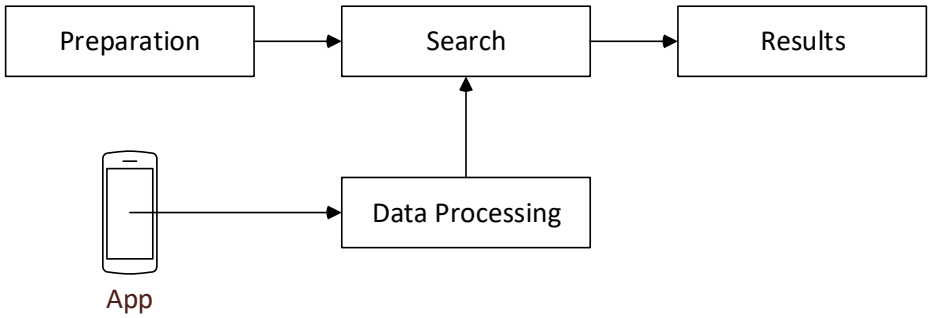
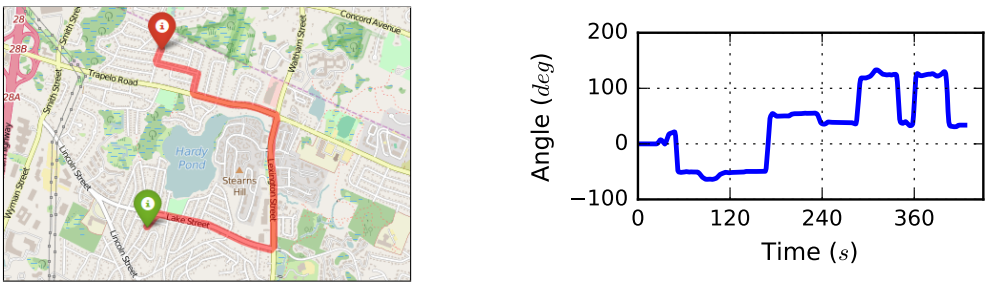


Figure 3.2: Block diagram of the location tracking attack.

- *Sensor Data Collection:* Sensor data is recorded by the app and sent to the colluding server. This step uses movement detection based on Accelerometer data to trigger sensor recording exclusively during vehicle movement.
- *Data Processing:* On receiving the sensor data, the server analyzes the data to derive the victim's trace of turn angles, curvatures, heading, accelerations and timestamps.
- *Search:* The search algorithm is run on the processed data and a ranked list of matching routes is produced.

Sensor data provides important information about a victim's movements. Among the three sensors, the Gyroscope sensor is most useful for this attack because of the following reasons: (a) The Gyroscope reports more accurate data than the others; (b) The Gyroscope reveals turn angles and road curvature of the undertaken route which are nearly static attributes and traceable on a public map resource. We heavily weight the Gyroscope data in this attack as the Accelerometer and Magnetometer strongly depend on dynamic factors such as traffic / road conditions or proximate magnetic fields, which are challenging to predict. Timestamps, Accelerometer and Magnetometer readings are used as supporting data to reduce noise and refine the results.

Data received from the Gyroscope is a sequence of three dimensional vectors reporting the rate of angular change along the victim's trajectory. Figure 3.3 illustrates an example of an experimental route and the corresponding angle sequence (processed from Gyroscope data) relative to the initial heading. Here, large changes in the angle trace



(a) Experimental route contains 6 turns from Start (green) to Stop (red).

(b) Angle trace contains 6 slopes (turns) and a few slight variations (curves).

Figure 3.3: An experimental route and the angle trace derived from the Gyroscope.

indicate turns at intersections. Right and left turns are represented by negative and positive slopes, while minor variations (e.g., less than 30° in the example) in between are attributed to road curvature.

We transform the Sensor-based Tracking Attack (Definition 1) to the problem of matching this angle trace and curvature with possible routes. The objective is to identify sequences of intersections and curvatures that match the slope change found in the angle trace. Our approach consists of graph construction based on OpenStreetMap, a public map resource, and matching routes on this graph with the actual angle trace using techniques similar to trellis codes decoding [75]. Note that in our context, the graph size is many orders of magnitude larger than typical trellis codes used in communications. In addition, while trellis codes make transitions and produce an output at each state, the victim's trajectory may traverse any number of atomic parts (transitions) without making a turn (output), rendering the problem more complex.

3.3 System Design and Algorithms

In this section, we describe our technique for constructing graphs for location tracking using sensor data, a basic search algorithm and scoring scheme using just Gyroscope turn information, and then a more advanced algorithm and filtering scheme using other sensor data. We also describe the error compensation and trace extraction steps performed on the sensor data to improve the search results.

3.3.1 Graph Construction

Our search is performed on a directed graph structure. For the sake of clarity, we first introduce some new definitions. Consider a geographic area $\mathcal{G} = (\mathcal{B}, \mathcal{C}, \theta, \vartheta)$. We assert that a connection between two atomic parts is a *non-turn connection* if the turn angle at the connection is below a threshold ϕ_{g3} (e.g., $\phi_{g3} = 30^\circ$, cf. Section 3.3.5). In this graph construction, we are interested in identifying such connections that can connect atomic parts together to create straight or curvy roads without including significantly large turns. We call such sequence of non-turn connected atomic parts a *road segment* (or simply *segment*). Specifically, a sequence $\vec{s} = (r_1, \ldots, r_l)$, where $r_i \in \mathcal{B}$, is a road segment if $\theta(r_i, r_{i+1}) \leq \phi_{g3}$ for $i = 1, \ldots, l-1$. Intuitively, a segment is a route without large turns at connections between its atomic parts. Additionally, we call segment $\vec{s}$ a *maximal-length segment*[1] if no atomic part can be added to $\vec{s}$ to form a longer segment while still preserving the non-turn condition. When a connection between two atomic parts has a turn angle greater than ϕ_{g3}, it becomes a connection between two segments, i.e., if $r \in \vec{s}, r' \in \vec{s}'$ and $\chi = (r, r') \in \mathcal{C}$, then $\theta(r, r') > \phi_{g3}$. In this case, we call χ a *segment connection* or simply an *intersection*.

Our idea for constructing the directed graph $G = (V, E)$ is to represent each segment $\vec{s}$ by a vertex $v \in V$ and each segment connection χ by an edge $e \in E$. An example construction is illustrated in Figure 3.1. Intuitively, one will stay at one vertex on the graph as long as they do not turn into another segment. A turn at an intersection makes them traverse to another vertex through an edge connecting them. Based on the public map resource, we accordingly build our graph for the whole geographic area. For each edge e corresponding to segment connection χ, we use $\theta(\chi)$ as the edge's weight. The length, speed limit, and curvature of a road segment $\vec{s}$ are stored as attributes of the corresponding vertex v. This information combined with the sensor data is used to match the victim's angle trace during the search. We note that for any two segments $\vec{s}$ and $\vec{s}'$ such that $\vec{s}' \subset \vec{s}$ (i.e., one is a sub-sequence of the another), we simply remove $\vec{s}'$ from the graph, because any atomic part r and connection χ of $\vec{s}'$ involved in the route search

[1] Maximal-length segment is analogous to a longest route between two nodes with an additional condition: weight (turn angle) must be small.

are also present in $\vec{s}$, rendering $\vec{s}'$ redundant. Therefore, graph G essentially contains only vertices corresponding to maximal-length segments, resulting in more efficient route search with greatly reduced graph size.

3.3.2 Basic Search Algorithm

The search technique includes maintaining a list of scored candidate victim routes while traversing the graph. When the search completes, a list of candidates is returned with their evaluated score. These routes have higher probability of matching the recorded mobility trace. For the current discussion, we assume that the adversary only exploits the Gyroscope data to launch the attack, i.e., we consider only g_t from $\mathcal{D} = \{(a_t, g_t, m_t)\}$. Let $\vec{\alpha} = (\alpha_1, \ldots, \alpha_N)$ be the derived sequence of turn angles at N intersections after processing the Gyroscope data g_t. The details of sensor data processing are discussed in Section 3.3.5. In Sections 3.3.3 and 3.3.4, we refine the algorithm and improve the performance by adding filtering rules and applying a more complex scoring method. Our goal at the moment is to find $\vec{\theta} = (\theta_1, \ldots, \theta_N) \in G$, the potential sequences of turns that maximize the probability of matching $\vec{\theta}$ given the observation of $\vec{\alpha}$. This probability, denoted $P(\vec{\theta}|\vec{\alpha})$, can be rewritten as:

$$P(\vec{\theta}|\vec{\alpha}) = \frac{P(\vec{\theta}, \vec{\alpha})}{P(\vec{\alpha})} = \frac{P(\vec{\alpha}|\vec{\theta})P(\vec{\theta})}{P(\vec{\alpha})}$$

As $P(\vec{\alpha})$ is the probability of a measurement $\vec{\alpha}$ without conditioning on $\vec{\theta}$, it is independent of $\vec{\theta}$. Thus, maximizing $P(\vec{\theta}|\vec{\alpha})$ is equivalent to maximizing $P(\vec{\alpha}|\vec{\theta})P(\vec{\theta})$. The distribution of a priori probability $P(\vec{\theta})$ may depend on the driver, city, and day / time of travel (e.g., home-to-work and work-to-home routes during weekdays have significantly higher probability than other routes). Since our goal is to demonstrate the generality of the attack even if the adversary knows nothing about the victim's travel history, we consider $P(\vec{\theta})$ to be equiprobable, i.e., any route has the same probability of being taken by the victim. This presents the worst-case attack scenario and gives a lower bound on the performance. If the a priori probability $P(\vec{\theta})$ is known, we expect the attack to achieve higher success probability than the performance we report in this work. Under the assumption of equiprobable a priori probability, the goal of maximizing $P(\vec{\alpha}|\vec{\theta})P(\vec{\theta})$ is equivalent to maximizing the probability $P(\vec{\alpha}|\vec{\theta})$ alone.

Input: $G = (V, E)$, $\alpha_1, \ldots, \alpha_N$

Output: U_N

Initialization: $U_0 \leftarrow V$; $U_1 \leftarrow \emptyset$; $\ldots U_N \leftarrow \emptyset$;
for $k = 1$ **to** N **do**
  for $u \in U_{k-1}$ **do**
    for $v \in V$ *such that* $(u, v) \in E$ **do**
      if filter(u, v, α_k) *passed* **then**
        $v.score \leftarrow u.score +$ scoring(u, v, α_k);
        $v.prev \leftarrow u$;
        $U_k \leftarrow U_k \cup \{v\}$;
      end
    end
  end
  $U_k \leftarrow$ pick_top(U_k);
end

Algorithm 1: Search Algorithm

Samples taken from the Gyroscope include noise as an additional unknown amount in the angle trace, yielding the angle $\vec{\alpha} = \vec{\theta} + \vec{n}$, where $\vec{n}$ is the random noise vector. We will show through experimental results in Section 3.4.1, that the Gyroscope noise can be approximated by a N-dimensional zero-mean normal distribution $\mathcal{N}(0, \sigma)$ with standard deviation σ. Accordingly, $P(\vec{\alpha}|\vec{\theta})$ can be rewritten as:

$$P(\vec{\alpha}|\vec{\theta}) = P(\vec{n} = \vec{\alpha} - \vec{\theta}) = \left(2\pi\sigma^2\right)^{-\frac{N}{2}} \exp\left(-\frac{\|\vec{\alpha} - \vec{\theta}\|^2}{2\sigma^2}\right)$$

where $\|\cdot\|$ indicates the L_2 norm of a vector. As $\left(2\pi\sigma^2\right)^{-\frac{N}{2}}$ is constant for a fixed N and σ, maximizing $P(\vec{\alpha}|\vec{\theta})$ is now equivalent to minimizing $\|\vec{\alpha} - \vec{\theta}\|$. Therefore, the adversary obtains the optimal solution as stated in Theorem 1.

Theorem 1. *Given graph G and a turn angle trace $\vec{\alpha}$ with normally distributed noise, the optimal route tracking solution is $\vec{\theta}^* = \arg\min_{\theta \in G} \|\vec{\alpha} - \vec{\theta}\|$.*

Based on Theorem 1, our search algorithm (Algorithm 1) aims at finding $\vec{\theta}$ that minimizes

$\|\vec{\alpha} - \vec{\theta}\|$. The main idea is to maintain a list of potential vertices (i.e., road segments) from which we develop the possible routes. The algorithm takes as input the graph $G = (V, E)$ and a sequence $(\alpha_1, \ldots, \alpha_N)$. The search consists of N rounds corresponding to a trace of N intersections. While the algorithm is similar to trellis codes decoding techniques in which paths are built up, maintained or eliminated according to a metric, our search is improved by filtering routes based on specific selection rules and keeping only top candidate routes after a number of iterations.

The algorithm starts by considering all vertices of the graph as potential starting points (initialization $U_0 \leftarrow V$). In each k-th round, we build a new list U_k of potential vertices as follows. For each vertex $u \in U_{k-1}$, we explore all its outgoing edges (u, v). During this traversal (line 4 – 10), filtering is applied (line 5) to eliminate such vertices / segments whose corresponding map data deviates too much from the actual sensor data. In this basic algorithm, the filter checks if the turn angle (i.e., the edge weight) between the current vertex u and candidate vertex v is within a specific range of the actual turn α_k. An edge (u, v) passes the filter only if $|\theta(u, v) - \alpha_k| \leq \gamma$, in which case v is put into U_k as a candidate for the next search iteration (line 8). The threshold γ depends on the quality of sensor data and is evaluated in Section 3.4.1. Note that when a vertex v does not satisfy the filtering rules, it simply means v is not used as a starting point in the next iteration, but may appear again if other starting points connecting to v satisfy the conditions.

At the same time when filtering is passed, the edge (u, v) is also evaluated for the likelihood to match the actual trace by the scoring function (line 6). The score for each k-th turn is computed by

$$\text{scoring}(u, v, \alpha_k) = d(\alpha_k, \theta(u, v)) = |\alpha_k - \theta(u, v)|, \tag{3.1}$$

where we compute the angle distance based on L_1 norm instead of L_2 norm for two main reasons: (a) computing L_1 norm requires less overhead; (b) in practice, we observe that L_1-based matching generally outperforms L_2-based, because Gyroscope errors are usually small, allowing L_1-based estimation to better overcome sparse large errors while L_2 norm tends to amplify such errors. The score of every route is initialized to 0 (line 1) and evolves to $\sum_{k=1}^{N} d(\theta(u, v), \alpha_k)$ after N iterations. When updating the score, we

additionally store the previous vertex ($v.prev$) of the candidate in order to trace back the full route (without storing the whole route) at the end of the search. We also note that as the list of candidates is developed through each iteration with non-negative metric, finding the actual route with loops is possible, because loops simply increase the score and are treated as regular routes (i.e., the search will terminate).

Since routes with lower score have higher matching probability $P(\vec{\alpha}|\vec{\theta})$, we only keep the top K candidates at the end of every iteration by calling pick_top function (line 12). It is noted that depending on attack configuration, pick_top may shorten the list of candidates only after some specific round. At the end of the search, based on U_N and previous vertex information stored for each candidate, the outcome $\mathcal{P} = \{\hat{\mathcal{R}}_1, \ldots, \hat{\mathcal{R}}_K\}$ is appropriately produced and returned.

Effect of Filtering and Top Selection: While scoring gradually distinguishes routes from each other, filtering can immediately eliminate a route at an early stage that can not be recovered later. There is a trade-off when determining the filtering thresholds. A tight rule can reduce the search time but may result in pruning more good routes due to early errors, whereas loose criteria reduces false elimination rate but increases running time and memory consumption. Similarly, selecting top candidates after some specific iterations can decrease the search time yet potentially removes good candidates that are bad at early stages. We leave the rigorous analysis of such parameters as future work. Instead, based on simulations and real driving experiments, we select appropriate parameters with respect to both attack performance and computation constraints such as memory and timing requirements.

3.3.3 Advanced Algorithm & Scoring Metrics

Algorithm 1 illustrates the main idea of our search technique, however, it represents a baseline attack as it relies only on the sequence of observed turn angles as the single input source to the algorithm. We now incorporate, into the basic search algorithm, the curvature of the undertaken route and the travel time between turns.

Curve Similarity: We define the curvature of the route as a sequence of angles between

intersections. Consider the victim's travel between the k-th and $(k+1)$-th intersections and let $T_k\delta$ (δ is the sampling period, and $T_k = 1, 2, \ldots$) be the victim's travel time for that distance. The curvature is then expressed by $\vec{C}_k = (\alpha_{k,1}, \ldots, \alpha_{k,T_k})$, where $\alpha_{k,i}$ are instantaneous directions at sampling time $i\delta$ on the k-th curve.

In order to match the sampled curvature with a candidate curve, we assume that the vehicle movement along the curve is at constant speed. Since no available data can provide sufficient accuracy for the instantaneous vehicular velocity, finding the best curve fit is challenging. The constant speed approach simplifies the estimation and greatly decreases the computation burden for each route. Our evaluation shows that curve matching with constant speed assumption considerably improves the attack performance. Specifically, we compute the angle sequence on each candidate curve as follows. For a candidate segment corresponding to a vertex u (which is either straight or curvy), we divide it into T_k equal-length sub-segments and consider each sub-segment as a straight line, then we find the orientations of sub-segments based on their geographic coordinates. This gives us $\vec{\vartheta}_u = \vartheta(u) = (\vartheta_{u,1}, \ldots, \vartheta_{u,T_k})$ as the curvature of u, where $\vartheta_{u,i}$ is the orientation of the i-th sub-segment.

Our goal is to maximize the probability $P(\vec{\vartheta}_u|\vec{C}_k)$ of matching a candidate curve $\vec{\vartheta}_u$ given the victim's curve $\vec{C}_k$ observed by the adversary. Due to the assumption of victim route equiprobability, as discussed previously in Section 3.3.2, we instead search for $\vec{\vartheta}_u$ values that maximizes

$$P(\vec{C}_k|\vec{\vartheta}_u) = P(\vec{n} = \vec{C}_k - \vec{\vartheta}_u) = \left(2\pi\sigma^2\right)^{-\frac{T_k}{2}} \exp\left(-\frac{\|\vec{C}_k - \vec{\vartheta}_u\|^2}{2\sigma^2}\right)$$

where $\vec{n} \leftarrow \mathcal{N}(0, \sigma)$ is the normally distributed random vector approximating the Gyroscope noise. We determine the curve similarity by

$$d(\vec{C}_k, \vec{\vartheta}_u) = \frac{1}{T_k} \sum_{i=1}^{T_k} |\alpha_{k,i} - \vartheta_{k,i}|.$$

Note that curve similarity, different from turn scoring in Equation (3.1), is normalized to mitigate the effect of bias scoring due to error accumulation on long curves (large T_k).

Travel Time Similarity: The tracking of the actual route based on turn angles and curvature information so far does not take into account the time scale of the victim's travel on

each road segment. To incorporate this information in the attack, we extract from Nokia's HERE map [76] the maximum allowed speed for every road in the geographic area $\mathcal{G}$ and compute the minimum time required to travel from one intersection to another along each road segment. Let $t_k \in \mathcal{D}$ be the actual time spent by the victim to travel from the k-th to the $(k+1)$-th intersection, and $\tau(u, v)$ be the minimum required time (computed from speed limit) for traveling from the last intersection to the current intersection (u, v) on the candidate route. The metric for the travel time similarity is computed by

$$d(t_k, \tau(u, v)) = |t_k - \tau(u, v)|.$$

Final Scoring Function: By incorporating the likelihood of the turn angles, the curvature, and the travel time along the search route, our final scoring function becomes scoring$(u, v, \alpha_k, t_k, \vec{C}_k)$ and is computed as

$$\omega_A d(\alpha_k, \theta(u, v)) + \omega_T d(t_k, \tau(u, v)) + \omega_C d(\vec{C}_k, \vec{\vartheta}_u) \tag{3.2}$$

where different weights $\omega_A, \omega_T, \omega_C$ can be selected dependently on the geographic area.

3.3.4 Filtering Rules

The search based on Gyroscope data is unaware of the absolute orientation of the routes. To refine the results and reduce the search time, we exploit the heading information derived from the Magnetometer to immediately eliminate bad routes (e.g., east-west routes are filtered out when the actual trace indicates north-south direction). In addition, we exploit the Accelerometer to identify idle states and discard samples in such periods for better estimation. We also extract speed information for each road and filter out routes by comparing the actual travel time between intersections with the time estimated for the segment under investigation.

Heading Check: At the time of each turn at an intersection, we extract the heading of the vehicle from the Magnetometer sensor sample and check that the next segment's direction should be close to the heading direction after turning. In practice, we observe that the Magnetometer may be influenced by an external magnetic field and the heading derived from the Magnetometer is not always accurate.

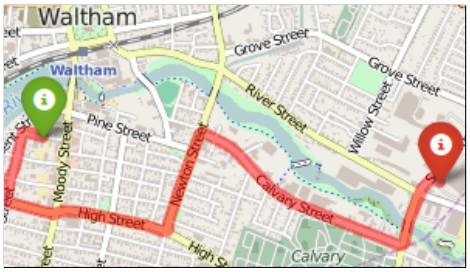
In order to exploit this information properly, we first verify that the Magnetometer data is reliable. When reliable, the magnitude of the heading vector will be within a certain range of values based on the geographic area $\mathcal{G}$. Specifically, the reliability is established if $M_l \leq \|m_t\| \leq M_h$, where $m_t \in \mathcal{D}$ is the Magnetometer vector, and M_l, M_h are lower and upper bounds that depend on $\mathcal{G}$. Also, most acceleration will be reported on the Accelerometer's y axis indicating that the device points in the vehicle's direction. Only after the reliability is assured, the orientation check is performed. The heading check is satisfied if $|h_k - \vartheta_{v,1}| \leq \phi_m$, where h_k denotes the heading vector of the vehicle after turning at the k-th intersection, $\vartheta_{v,1}$ is the orientation of the first sub-segment of segment v connected to u, and ϕ_m is the Magnetometer error threshold. Note that in case of unreliable Magnetometer data, the check is not performed but v is not eliminated.

Travel Time Check: Due to the maximum speed limit on each road, the travel time cannot be arbitrarily small. Given the actual travel time duration $t_k \in \mathcal{D}$ between the k-th and $(k+1)$-th intersections, the maximum distance traveled by the vehicle is $L_k \leq L_{\max} = \beta V_{\max} t_k$, where $V_{\max}$ is the regulated speed limit, and $\beta \geq 1$ is the over-speeding ratio that can be reached by the vehicle. Consequently, during the search we only keep such candidate routes that are not longer than $L_{\max}$. To reduce the computation overhead, we instead precompute $t_v = \frac{L_v}{V_{\max}}$ for each candidate road segment v of length L_v, and our timing rule becomes $t_k \geq \frac{t_v}{\beta}$, i.e., $L_v \leq L_{\max}$. We emphasize that in realistic scenarios, since the vehicle may drive at any speed below the limit or may get stuck in the traffic for an unpredictable duration, the travel distance can be arbitrarily small. Therefore, no non-zero lower bound on segment length is established.

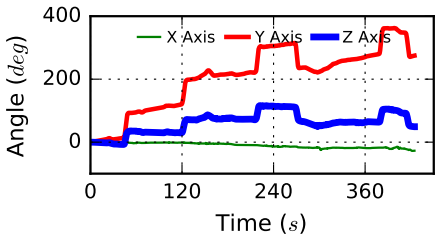
3.3.5 Sensor Data Processing

A big challenge in implementing this attack is extracting accurate route information from noisy sensor data. Along with the external factors discussed before (e.g., potholes, bumps, road slopes, magnetic field and driver behavior), some internal misconfiguration may also introduce errors in the data.

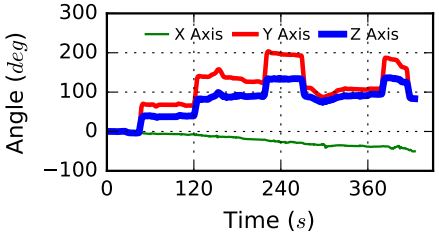
Axis Misalignment: Sensor x, y and z axes may not have perfect orthogonal alignment. This causes a bias in the sensor values which can be defined as the deviation from the



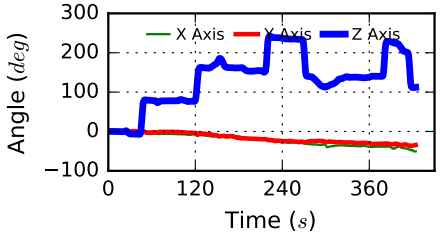
(a) Experimental route



(b) Recorded Gyroscope data



(c) Calibrated Gyroscope data



(d) Rotated Gyroscope data

Figure 3.4: Error compensation steps for Gyroscope data for an experimental route.

expected x, y and z values when the device is at rest. The bias can typically be removed by subtracting them from the reported x, y and z sensor values.

Thermal Noise: The sensor's x, y and z axes values may also vary with the device / sensor temperature. Some Operating Systems compensate for this noise by pre-filtering the data, but at the cost of reduced accuracy.

Given these errors, we decompose the sensor data processing into error compensation and trace extraction tasks.

Error Compensation: Error compensation consists of a calibration phase followed by rotation of the data. Note that while our discussion focuses on Gyroscope data, similar tasks can be performed for the Accelerometer and Magnetometer.

Calibration: The Gyroscope sensor bias and vehicle vibration result in angle drift, i.e., the values change linearly[2] in time even at idle. An example of experimental route is shown in Figure 3.4a. The Gyroscope data is reported as a sequence of angle change

[2]Our observation suggests linear model well approximates the angle drift.

between sampling periods, and integrated over time to obtain the relative (with respect to the initial recording) angle sequence in x, y, and z axes shown in Figure 3.4b. For the experimental route, the resulting angle sequence shows a large positive drift in the y axis. To compensate for the drift, we assume the vehicle is in a parked state in the calibration phase (we note that this is only required once for subsequent attacks). The drift vector is estimated as $\overline{\Delta\alpha} = \mathbf{E}[\Delta\alpha/\Delta t]$, the expected angle change rate. The calibration is then performed by subtracting $\overline{\Delta\alpha}$ from the angle sequence (Figure 3.4c). Note that complete removal of drift is a difficult task and would require more computation-expensive mechanisms, e.g., *Sensor Fusion* algorithms.

Rotation: Recall that a victim can place their smartphone in any orientation in the vehicle. To simplify the attack computation, we rotate the sensor data to a reference coordinate system, where the x axis points from left to right of the driver, the y axis aligns with the heading direction of the vehicle, and the z axis points upward perpendicularly to the Earth surface. After rotation, the x and y values are then used to measure *pitch* and *roll* respectively, while turn angle information is indicated in the z axis (Figure 3.4d).

Trace Extraction: The rotated Gyroscope data contains all the turn and curvature information in the z axis. We use these z values of the Gyroscope data to extract the victim's turn angles at intersections and curves between them. The acceleration vectors are used to improve the search performance by detecting vehicle's idle states.

Turn and Curve Detection: The left and right turns of a route are distinguished according to positive and negative angle changes in the rotated data. Our idea for identifying intersections is illustrated in Figure 3.4d, where left turns are identified by an increasing slope within a short period of time and right turns correspond to decreasing slope. More precisely, let z_i be the Gyroscope value on the z axis at time $i\delta$ in the rotated angle trace. An intersection is found if it satisfies *all* the following conditions:

1. **Start turn:** The angle change between time $i\delta$ and $(i+1)\delta$ is higher than a threshold ϕ_{g1}, i.e., $|z_{i+1} - z_i| > \phi_{g1}$, which captures the event that the vehicle is starting to make a turn or enter a curve.

Table 3.1: Default parameters used in evaluation of the location tracking system.

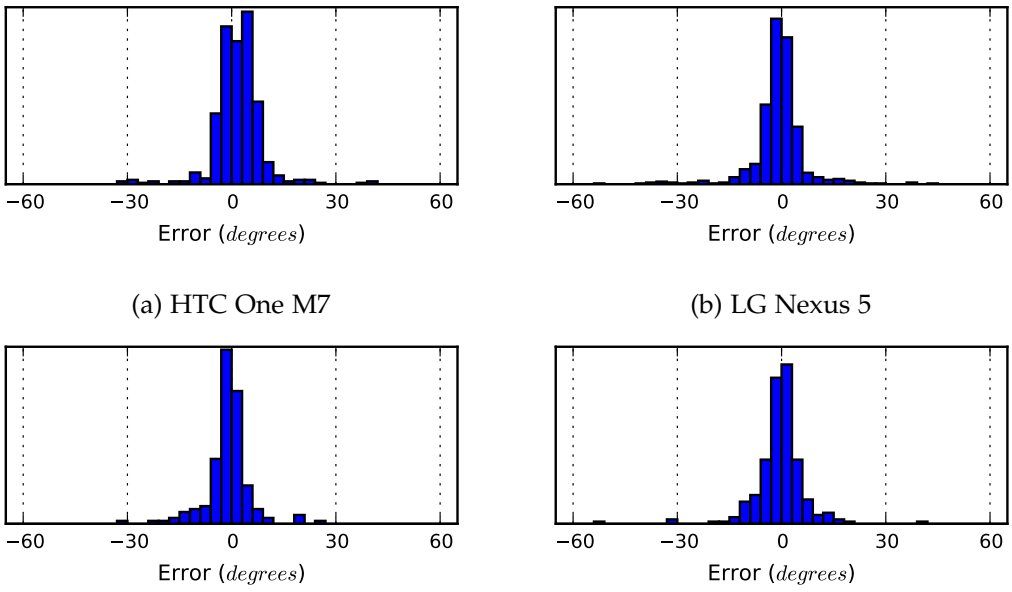
Parameter	Value
Scoring weights	$\omega_A = 2.5, \omega_T = 0.1, \omega_C = 2.5$
Turn/curve detection threshold	$\phi_{g1} = 1°, \phi_{g2} = 10°, \phi_{g3} = 30°$
Turn angle filtering threshold	$\gamma = 60°$
Heading filtering threshold	$\phi_m = 90°$
Travel time filtering threshold	$\beta = 1.5$
Noise distribution	$\mu = 0.003, \sigma = 7.54$
Sampling period	$\delta = 100\,\text{ms}$
Top candidates limit	$K = 5000$, for iterations $k \geq 2$

2. **Large deviation:** The largest deviation on a slope under investigation must be greater than a threshold ϕ_{g2}, i.e., $\max_{i \in \text{slope}} |z_{i+1} - z_i| > \phi_{g2}$. This distinguishes the real turn from a slight curve on the route.

3. **Large turn angle:** If the difference between the first and the last angle on a slope is greater than ϕ_{g3}, i.e., $|z_{i+n} - z_i| > \phi_{g3}$, the slope is recognized as a real turn, and the value $\alpha_k = z_{i+n} - z_i$ is the turn angle for the corresponding k-th intersection.

A curve is recognized if the first condition is met, but the other two conditions do not hold at the same time. In all other cases, the road segment under investigation is considered a straight segment. The parameters ϕ_{g1}, ϕ_{g2}, and ϕ_{g3} are configured accordingly to the geographic area.

Idle State Detection: Despite the limited accuracy of the Accelerometer to reveal the precise instantaneous vehicular speed, we can still exploit it to differentiate an idle state (e.g., vehicle stops at traffic lights) from movement on a straight road. In both cases, the Gyroscope does not expose large enough variations for detecting angle changes with adequate accuracy. However, the former case results in nearly zero magnitudes of acceleration vectors on the Accelerometer, while the values are considerably larger with higher fluctuations for the latter case. The idle states, therefore, can be extracted as blocks of time that have near zero Accelerometer magnitudes. With idle states detected, we can better estimate the actual non-idle time and improve the attack performance.



(a) HTC One M7

(b) LG Nexus 5

(c) LG Nexus 5X

(d) Samsung S6

Figure 3.5: Gyroscope noise distributions as measured in real driving experiments for different smartphones.

3.4 Evaluation

In this section, we evaluate the attack efficiency based on simulations and real driving experiments. First, we justify the accuracy of the Gyroscope sensor and present our selection criteria for cities chosen for evaluation. Subsequently, we present our simulation and real driving results with a discussion on attack performance and its implications on user privacy. The attack parameters with default values are given in Table 3.1.

3.4.1 Accuracy of the Gyroscope

The Gyroscope sensor is less impacted by the environment in comparison to the Accelerometer and Magnetometer. The sensor is also heavily weighted in our attack. Therefore, it is important to first justify the accuracy of Gyroscope data by measuring the turn errors based on real driving experiments. We use 4 smartphones of different brands and models, and take total 70 driving routes in both Boston and Waltham (Massachusetts, USA). To assess the Gyroscope errors, we extract the truth turn angles θ_i of taken routes

Table 3.2: List of phones tested for accuracy along with the number of turns, and the Gyroscope noise's mean and standard deviation.

Phone	**Turns** N	**Mean** μ	**Std. dev.** σ
HTC One M7	482	1.73°	7.07°
LG Nexus 5	618	-0.77°	7.89°
LG Nexus 5X	170	-1.12°	6.40°
Samsung S6	238	-0.57°	7.51°

from OpenStreetMap, then for each θ_i, we obtain the Gyroscope angle α_i (after sensor data processing phase) and compute turn errors $e_i = \alpha_i - \theta_i$. As observed from Figure 3.5 showing histogram of e_i, the error distribution for each phone closely follows a *normal distribution* with more than 95% of errors below 10°. Table 3.2 indicates almost equal noise standard deviation of each device. For all routes combined for 4 phones, the mean μ and standard deviation σ values are 0.003 and 7.54, respectively.

3.4.2 Evaluation of Simulation Routes

Selection of Cities: To assess the attack's impact on diverse cities of the world, we identified 11 cities for simulations based on their size, density and road structure. Table 3.3 summarizes their attack-related characteristics such as the graph size (number of vertices $|V|$ and edges $|E|$) and distribution of turn angles at intersections (mean μ_{turn} and standard deviation σ_{turn}).

Big cities such as Atlanta, Boston, London, Madrid, Paris, and Rome create larger graphs than the rest according to our construction method. While Manhattan is quite populated, it has the smallest graph in our set because the graph only contains maximal-length segments. Manhattan is dominated by long east-west and north-south roads, many of which are parallel forming fewer segments. The top cities with grid-like road structures are Atlanta, Manhattan, and Sunnyvale with low values of σ_{turn}. Berlin, Boston, London, and Waltham have more spread out turns, but not as much as Madrid, Paris, Rome, and Concord. Figure 3.6 shows the turn angle distributions for some selected cities, where we observe that the majority of intersections in Sunnyvale and Manhattan are 90° while the others have more unique turns.

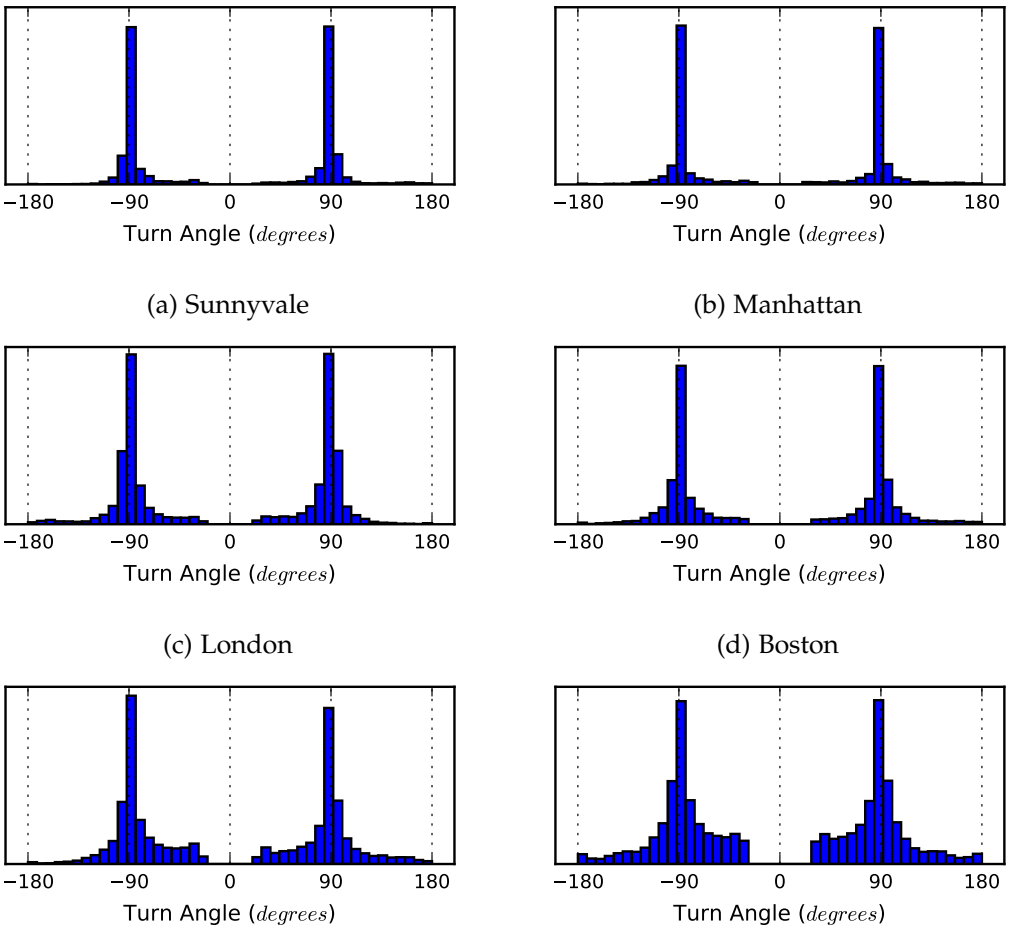
Table 3.3: List of cities used for the location tracking attack evaluation with their characteristics: graph size ($|V|, |E|$) and turn angle distribution ($\mu_{\text{turn}}, \sigma_{\text{turn}}$).

City	|V|	|E|	Mean μ_{turn}	Std Dev σ_{turn}
Atlanta, GA, USA	10529	25557	88.73°	17.58°
Berlin, Germany	4708	19752	88.21°	19.87°
Boston, MA, USA	8010	22149	89.69°	20.52°
Concord, MA, USA	3049	6467	88.13°	29.58°
London, UK	9468	21968	87.83°	20.38°
Madrid, Spain	10012	30144	86.41°	25.13°
Manhattan, NY, USA	1033	3699	89.23°	17.81°
Paris, France	6744	11204	86.35°	26.26°
Rome, Italy	9408	20577	85.98°	26.15°
Sunnyvale, CA, USA	5592	12302	88.59°	16.00°
Waltham, MA, USA	3366	9437	88.93°	20.53°

Creation of Simulation Routes: We test the feasibility of the attack on selected cities by running the system on simulated routes. In case of Boston and Waltham, we also collected 140 driving routes used for experimental evaluation (cf. Section 3.4.3). Both sets of simulation and real routes are converted to the same format for compatibility, in which the user's route is represented as a sequence $\mathcal{U} = ((h_1, \alpha_1, t_1, \vec{C}_1), \ldots, (h_N, \alpha_N, t_N, \vec{C}_N))$. The heading vector h_i represents the direction of vehicle right before entering an intersection with turn angle α_i, whereas t_i and $\vec{C}_i$ are the time duration and curvature of the travel between the previous intersection and the next one.

Route Generation: Using the constructed graph $G_{\mathcal{G}} = (V, E)$ for a selected city $\mathcal{G}$, each simulation route is created by first randomly choosing a route length $N \leftarrow \{4, \ldots, 11\}$ and then adding N random connected segments that satisfy (a) turn angle constraint: $30^\circ \leq |\alpha_i| \leq 150^\circ$, (b) travel time constraint: $t_i \geq 10\,\text{s}$. Note that as these segments are maximal-length, the system may choose connections that are large distances apart for larger segments. In our simulations, the generated routes are between $\approx 0.5\,\text{km}$ and $\approx 48.15\,\text{km}$ with an average length of $\approx 7.15\,\text{km}$.

Adding Noise: To simulate realistic scenarios, we add various levels of noise to the



(a) Sunnyvale

(b) Manhattan

(c) London

(d) Boston

(e) Rome

(f) Concord

Figure 3.6: Distribution of intersection turn angles in selected cities.

route's characteristics. The Magnetometer noise n_m is added to h_i by a *uniform distribution* such that $-90^\circ \leq n_m \leq 90^\circ$. To mimic the travel time in practice, we add *uniform distributed* noise n_t to t_i such that $\frac{t_i}{\beta} \leq t_i + n_t \leq \frac{t_i}{\beta'}$, where β is the over-speeding ratio and β' is the lower bound speed ratio which attempts to model the slow driver or traffic jam. While β is fixed to 1.5, β' is varied depending on simulation scenarios defined shortly below. The Gyroscope noise is finally added to both turn angles α_i and curvature $\vec{C}_i$ according to a normal distribution $\mathcal{N}(\mu, \sigma)$ with $\mu = 0.003$ (obtained from Section 3.4.1). Note that the noise margin with simulated Magnetometer and travel time is relatively higher than in reality; for instance, the Magnetometer error is found to be only around

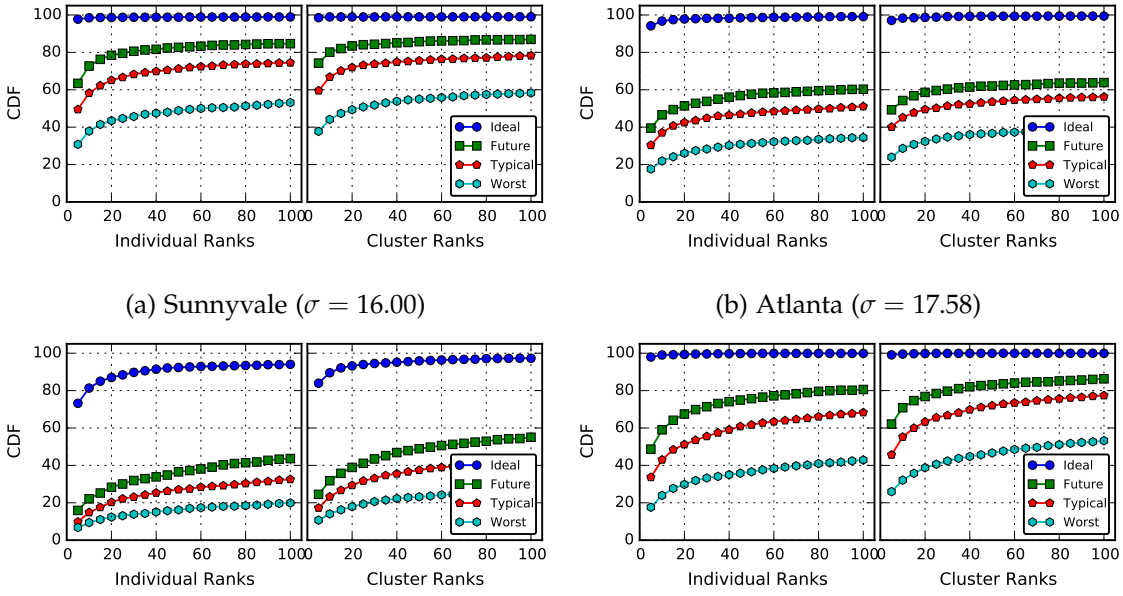
60° for our devices, while in practice drivers rarely exceed 15% (i.e., $\beta = 1.15$) of speed limit (e.g., 75 mph in a 65 mph speed zone in Boston).

Simulation Scenarios: To understand the attack performance under various environments, our simulation evaluation is performed and reported for different scenarios in which several noise parameters are adjusted from the above settings.

- **Ideal:** noise-free scenario (upper bound performance).
- **Worst:** $\sigma = 10, \beta' = 0.1$. In this scenario, we consider heavy traffic and old smartphones with less accuracy.
- **Typical:** $\sigma = 8, \beta' = 0.5$. In this scenario, we consider moderate traffic and current smartphones. Note that $\sigma = 8$ is slightly higher than the experimental value $\sigma = 7.54$, implying a slightly harder attack.
- **Future:** $\sigma = 6, \beta' = 0.5$. In this scenario, we consider moderate traffic and future smartphones equipped with more accurate sensors as MEMs technology progresses.

Simulation Results: We evaluate the potential of the attack for all cities in Table 3.3 using the above 4 noise scenarios. In total, there are 44 test cases and we generate a new set of 2000 simulation routes for each test case. We use the same scoring weights $\omega_A = 2.5, \omega_T = 0.1, \omega_C = 2.5$ for every city. These weights are selected as they are relatively good for all cities, and our main simulation goal is to evaluate the attack using the same configuration for different city profiles. Other parameters used for the attack are specified in Table 3.1. The attack outcome is evaluated according to both *individual rank* and *cluster rank*. For the latter metric, we choose the proximity threshold $\Delta = 500$ meters, which typically covers a few house blocks or apartment buildings.

Figures 3.7 and 3.8 shows the *Cumulative Distribution Function (CDF)* of individual and cluster ranks (i.e., P^{idv} and P^{clt}) produced by the attack. For the *Typical* scenario, we see that the system is able to find more than 50% (resp. 60%) of exact routes (resp. clusters of routes) in the top 10 results for all cities except for Atlanta, Berlin, and Manhattan. Even in the *Worst* scenario, more than 35% (resp. 40%) of exact routes (resp. clusters) are found in the top 10 results. In case of cluster rank, we examine the results in more detail and



(a) Sunnyvale ($\sigma = 16.00$)

(b) Atlanta ($\sigma = 17.58$)

(c) Manhattan ($\sigma = 17.81$)

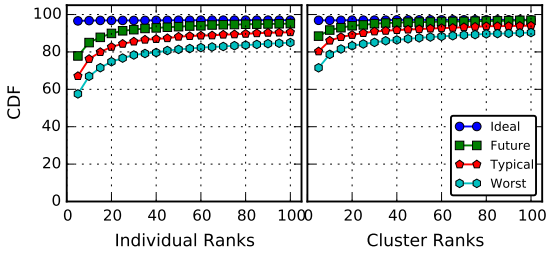
(d) Berlin ($\sigma = 19.87$)

Figure 3.7: Attack performance on simulation routes for cities with less unique turns (low σ_{turn}).

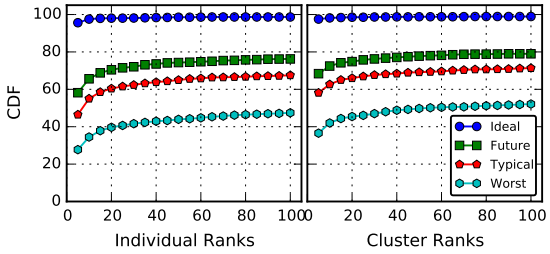
find that each cluster comprises of a small set of routes (approximately 1-20 routes per cluster). This explains why cluster ranks are only slightly better than individual ranks.

Among cities having low σ_{turn} (less unique turns) in Figure 3.7, Manhattan results in lower ranking than Atlanta and Sunnyvale even when it has a higher σ_{turn} and smaller graph size (lower $|V|$ and $|E|$). This can be attributed to two factors: (1) Manhattan has mostly straight roads reducing the curvature impact on scoring, and (2) most roads are parallel rendering heading filters ineffective. Atlanta and Sunnyvale, on the other hand, have more curvy roads that do not run in parallel. Atlanta has a lower ranking than Sunnyvale due to many more segments and connections that significantly increase the search space and inversely affect the results. Berlin, like others in this group, has many 90° turns and straighter roads and reports results in between Atlanta and Sunnyvale.

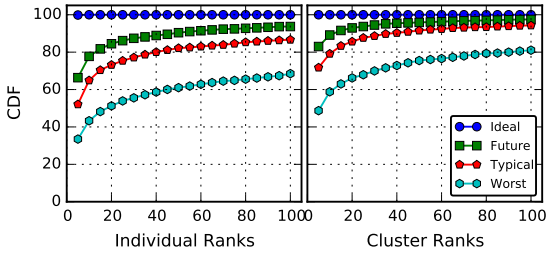
Among cities having high σ_{turn} (more unique turns) in Figure 3.8, the turn angle impact on scoring is high (especially very high for Madrid, Rome, Paris and Concord, cf. Table 3.3). The attack for Concord is most successful because its high number of



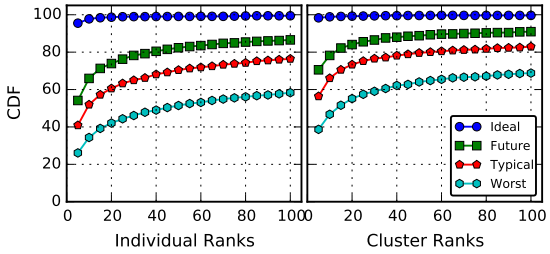
(a) London ($\sigma = 20.38$)



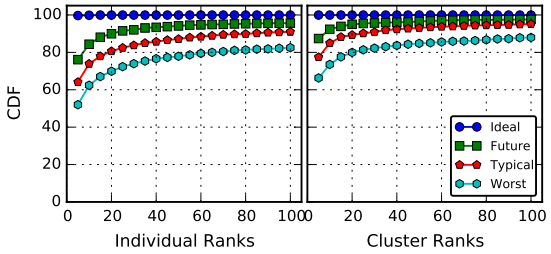
(b) Boston ($\sigma = 20.52$)



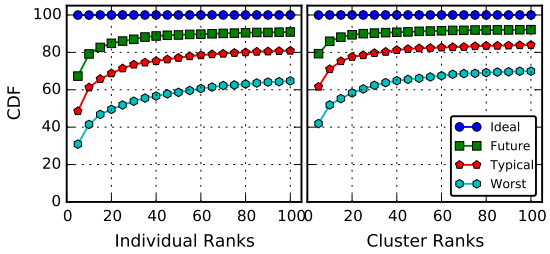
(c) Waltham ($\sigma = 20.53$)



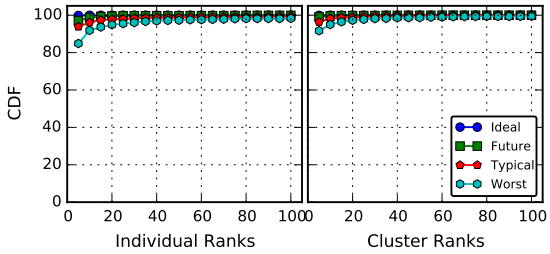
(d) Madrid ($\sigma = 25.13$)



(e) Rome ($\sigma = 26.15$)



(f) Paris ($\sigma = 26.26$)



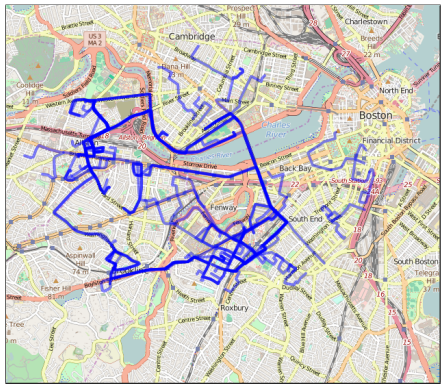
(g) Concord ($\sigma = 29.58$)

Figure 3.8: Attack performance on simulation routes for cities with more unique turns (high σ_{turn}).

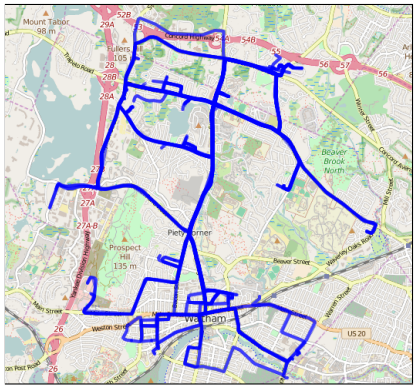
curvy roads and unique turns helps diversify the route's score, and the small graph size significantly reduces the search space. Paris creates somewhat more difficulty for the adversary in comparison to Rome and London even though it has a higher σ_{turn} and lower $|V|$ and $|E|$. This can be explained by the fact that many internal roads in Paris are straight, reducing the curvature impact on scoring. Madrid, like Paris, also has a lot of straight roads and the rankings are slightly lower than Paris due to a high $|V|$. The attack seems easy in Rome and London thanks to the high variations in curvature in both cities. Boston has lower ranking than London even when it is similar in turn distributions and graph size. This is mainly because Boston has several grid-like residential areas such as South Boston and Back Bay that create much confusions for routes passing through such areas. Waltham's road structure is similar to Boston's except that it is much smaller, which becomes the main factor for increasing the attack performance.

3.4.3 Evaluation of Real Driving Experiments

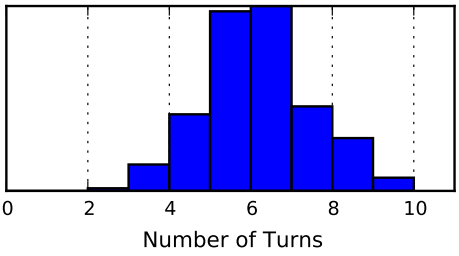
To measure the attack efficiency in reality, we collected real driving experiments in Boston and Waltham. For each city, over 70 different routes were driven. These routes emulated realistic scenarios, e.g., traveling between residential areas, shopping stores, office, or city centers. There were 4 drivers participating in the experiments and they were instructed to (1) place the phone anywhere but in fixed position during collection, (2) idle for at least 10 seconds before driving, and (3) drive within the city limit and take a minimum of 3 turns on their route. These requirements allow us to model typical realistic scenarios, in which the victim puts their phone in a stable position (cup holder, mount, etc.), then takes a few seconds to put on the seatbelt, and adjust the seat, mirrors, or lights. In this initial study, we did not consider situations when the vehicle starts by reversing. We emphasize that given the limited resources, we aimed to obtain a data-set as diverse as possible and did not request the drivers to repeat the same routes. Still, all routes consist of total $\approx 980\,\text{km}$, including driving in both peak and off-peak hours. Scoring weights $(\omega_A, \omega_T, \omega_C)$ were fine-tuned based on road characteristics: $(3.0, 0.1, 2.75)$ for Boston, and $(2.25, 0.1, 2.5)$ for Waltham. Boston has more unique turns than curves attributing to the higher ω_A, while Waltham has more unique curves than turns attributing to higher ω_C.



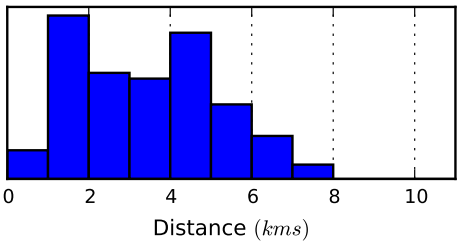
(a) Traveled routes in Boston



(b) Traveled routes in Waltham



(c) Turns Distribution



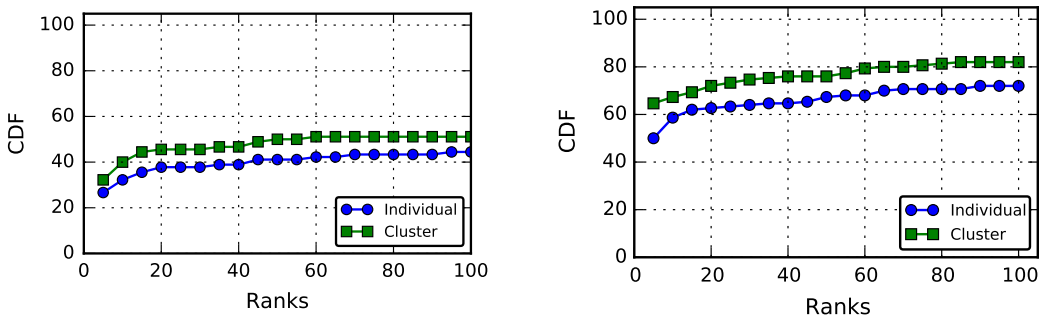
(d) Distance Distribution

Figure 3.9: Real driving experiments statistics showing the GPS traces for all traveled routes; and the Turn and Distance distributions for all routes combined.

Waltham has typically less traffic than Boston, therefore, we assign lower ω_A and ω_C to increase the impact of ω_T.

Figure 3.9 shows the distribution of turns made on all routes, total traveled distances, and the GPS traces. Note that GPS is used only for ground truth comparison. The shortest route driven was ≈ 0.75 km and the longest ≈ 7.25 km. Additionally, 4 more routes were driven to consider scenarios of driving in a circle, taking many turns (≥ 20), and traveling longer distances (≥ 20 km). These routes were also used to test the system's stability.

Figure 3.10 shows the attack performance in terms of both individual and cluster ranks. The reported results are a worst-case scenario with no a priori information on the user's routes. We see that roughly 60% of routes in Waltham and roughly 30% of routes in Boston are in the top 10 individual ranks. When top 1 is considered (i.e., exact route), the success probability reduces to 38% for Waltham and 13% for Boston, respectively.



(a) Boston

(b) Waltham

Figure 3.10: Attack performance on real driving experiments collected in Boston and Waltham.

The gap between individual and cluster ranks is about 10%, which is almost similar to simulations. The number of routes per cluster is around 2-3 for most top ranked clusters. The performance for both cities lies between the simulation's *Typical* and *Worst* scenarios. However, the results for Boston are closer to the *Worst* scenario, while Waltham's are much like the *Typical*. The main reason for this difference is the traffic in Boston that caused more variations in estimating non-idle time than Waltham. The small gap between real and simulation results shows that our simulation framework may serve as an effective model for studying the attack in a larger scale where experiments are limited.

3.4.4 Feasibility of the Attack

The colluding server was setup as a Linux Virtual Machine on a Dell PowerEdge R710 server. It had 2x4 cores with 16 threads running at 2.93 GHz, with 32 GB of RAM. The system is written in Python and run using *PyPy*, a fast Python JIT compiler. We measure the feasibility of the attack in terms of execution time for processing data and searching routes. The search time specifically depends on the route length and graph size.

Data Processing: The longest experimental route (approximately 45 minutes long) in our set requires ≈ 1.4 s to process the sensor data and produce a trace of heading, turns, curves, and timestamps, while an average route takes $0.1 - 0.2$s.

Route Search: The search algorithm has a worst case time complexity of $O(nve)$, where

Table 3.4: Test cases for impact of parameters and calibration.

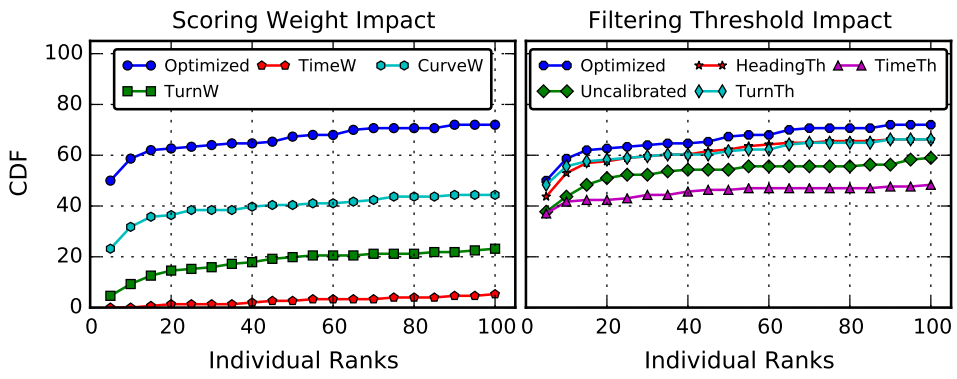
Test case	**Parameter settings**
Optimized	As in Section 3.4.3
TurnW	As *Optimized*, except $\omega_T = 0, \omega_C = 0$
TimeW	As *Optimized*, except $\omega_A = 0, \omega_C = 0$
CurveW	As *Optimized*, except $\omega_A = 0, \omega_T = 0$
HeadingTh	As *Optimized*, except $\phi_m = 30^\circ$
TimeTh	As *Optimized*, except $\beta = 1.0$
TurnTh	As *Optimized*, except $\gamma = 20^\circ$
Uncalibrated	*Optimized* without calibration

n denotes the number of turns in a route, v denotes the number of vertices in graph $G = (V, E)$, and e denotes the number of edges of v such that $(v, e) \in E$. The value of n is typically quite small for a route (< 10). The value of e is also not large and varies based on the length of maximal-length segments of the area. For Atlanta, the largest city in our set, this value ranges between 1 and 123, with a mean of 30. Note that this represents a worst case timing for our attack, and the top path selection and filtering rules significantly reduce the search space (cf. Sections 3.3.2 and 3.3.4). In reality, the search for each route in Atlanta takes only about 2.2 s. For Concord, the smallest one, each route takes about 0.4 s. We use 15 threads to parallelize the search on multiple routes, and 1 remaining thread for control and management. Using these settings, the simulation of 88000 routes took $\approx$ 21 hours to complete ($\approx$ 0.85 s per route).

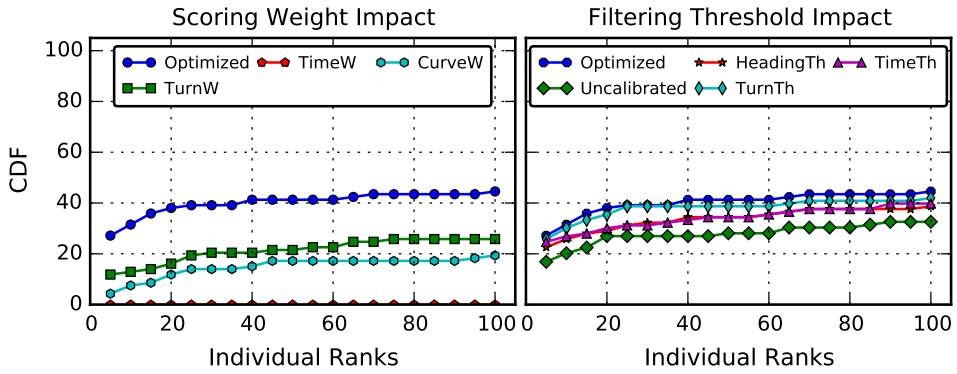
This indicates that the attack is practical (e.g., less than 4 seconds for a long route in Atlanta). With adequate resources, an adversary can quickly search millions of routes.

3.4.5 Impact of Algorithm Parameters on Attack Performance

In this subsection, we study the attack performance under various conditions such as when calibration is not performed, or the algorithm parameters are not carefully selected. We use the real driving experiments from Boston and Waltham and re-perform evaluation changing one parameter at a time to better understand the impact of individual parameters. For comparison, the performance achieved with parameters optimized



(a) Impact of changing parameters and calibration for Waltham experiments.



(b) Impact of changing parameters and calibration for Boston experiments.

Figure 3.11: Impact of changing parameters and calibration on the attack for real driving experiments.

in Section 3.4.3 is referred to as the *Optimized* test case (cf. Table 3.4 and Figure 3.11).

Scoring Weights: To justify the impact of each scoring weight, we ignore the other weights by setting them to zero in the scoring function, cf. Equation (3.2). Figure 3.11a shows that curvature is the most useful factor for success probability in Waltham, while turns only slightly increases the performance. This is not only applicable to Waltham, but also to cities that have numerous roads with unique curvature. Figure 3.11b shows that turns is somewhat more useful for success probability in Boston. The travel time varies more due to external factors such as traffic or unknown speed, making it less impactful. Hence, weights must be selected based on the target area to maximize the attack success.

Filtering Thresholds: Filtering allows quick elimination of bad routes, however, it can also falsely remove good routes. To see the performance impact from over-filtering, we reduce the thresholds for turn, heading, and travel time as specified in Table 3.4. We observe from Figure 3.11 that tighter turn, heading and travel time thresholds do not significantly decrease performance, which implies that the sensors have small noise margin and stricter rules can be applied to speed up the search if execution time is of high priority.

Calibration: Recall that drivers were instructed to stay idle for at least 10 s before driving. While this allows for easy calibration, an alternative calibration method can be used that first detects idle time (based on Accelerometer) and then computes the Gyroscope drift during that state. This enables calibration whenever the vehicle is idle (e.g., stopped at traffic lights) and the parking assumption can be relaxed. We observe from Figure 3.11 that performance does not decrease significantly even without calibration. The individual ranks drop by $10\% - 15\%$ in comparison with *Optimized* which implies calibration is optional rather than a required operation.

3.4.6 Impact of Assumptions on Attack Performance

In this subsection, we discuss the impact that the assumptions made for this work may have on the attack performance.

Route Equiprobability: We emphasize that the reported results in this work are based on the worst-case assumption of no a priori information of the victim's travel history. Knowing the starting or ending point would improve the accuracy. On the other hand, such travel history information can be built up over time to improve the attack. We plan to study such extensions in the future.

Fixed Position: Our assumption of fixed phone position is realistic in various scenarios (e.g., many states in the USA prohibit hand-held use). However, users may still interact with their phones while driving. We describe an idea (we did not implement it) that can increase the possibility of distinguishing between a real turn and a change in phone's orientation due to user interaction. The idea is based on the observation that human

interaction (e.g., touching, holding in hand) induces high variations in sensor data in *all* 3 dimensions for a short duration. Note that if the variations are low, the attack is barely affected and there is no need for detection. When such events are detected, we simply ignore the sensor data, and later, re-perform rotation to reflect the phone's new position. In practice, however, more complex algorithms would be required to deal with noise and unknown human behaviors, which can be studied in the future.

Detection of Vehicle Start: In this work, we assume that it is feasible to determine when a user enters their vehicle. This can be done a posteriori with the app continuously recording the sensors (storing a window of few minutes) and using techniques described in [77] to detect when the user stops walking and steps into the vehicle.

Reversing: In this work, we assume forward-only motion of drivers. While reversing can be detected using the Accelerometer, a more complex problem may arise when turning is performed at the same time as reversing (e.g., making a U-turn or parallel parking). This increases the search space, and our algorithm would have to be extended to roll back to previous states along all candidate routes.

Known City: Knowledge about the victim's city can be obtained in several ways. For instance, the app can detect the city based on IP address when the victim is connected to Wi-Fi or cellular networks. Additionally, an adversary with access to the victim's social network can find the victim's city, frequently visited places, and even route patterns. A powerful adversary can also run the attack on multiple geographic areas in parallel. These techniques can be combined together to devise an effective attack.

3.5 Related Work

Smartphone privacy attacks have recently attracted significant interest. They typically fall into one of three categories: (1) most attacks use cellular signals, GPS, Wi-Fi, Bluetooth, NFC, Wi-Fi Direct and other radio communication mechanisms (henceforth, we will refer to them as wireless location support systems or WLSS), (2) sensor centric attacks use native smartphone sensors such as the Gyroscope, Accelerometer and Magnetometer as data sources with no WLSS involvement, and (3) the hybrid cases are where the victim

makes available, albeit to a limited community and on a limited basis, their location. These attacks use WLSS and sensor data integration.

Fawaz et al. [2] reported that 85% of surveyed users expressed concern about conveying location information. Some countermeasures emerged in the form of location privacy protection mechanisms or LPPMs. These services obfuscate location information by modifying precision or performing location transformation. As they attempt to deflect WLSS centric threats, LPPMs remain ineffective in mitigating our threat.

WLSS Based Attacks: WLSS based attacks typically require either apps installed on a smartphone with appropriate permissions or significant presence within the network infrastructure. We do not address the former as the user consciously forfeited some degree of position anonymity. The infrastructure attack involves taking over some of the infrastructure components or injecting signature probes and are subject to detection by conventional means (i.e. IDS or IPS solutions). WLSS attacks provide accuracies near 90% when attempting path identification.

Qian et al. [78] attempted targeted cellular DoS attacks. Of relevance is identifying the specific smartphone location as a precursor to the attack. The attack seeks to gain IP identification using techniques like active probes and fingerprints. By measuring promotion delay and Round Trip Time (RTT), cellphone localization is achieved with granularity to the Location Area Code (LAC) / Radio Network Controller (RNC) range. Its effectiveness is limited due to measurement tuning needs and RNC sharing observed among smaller cities. This expands the geographical area cross section from which to identify the user. As with WLSS attacks, introducing network probes may enable detection.

Kune et al. [79] describe location determination via leakage from lower level Global System for Mobile Communications (GSM) broadcasts, in particular, a victim's temporary identifier. For this attack to work, the attacker must initiate a Paging Control Channel (PCCH) paging request targeting the victim and passively listen for broadcast PCCH messages. Although relatively simple, it places the attacker as an active network participant which risks detection. It also requires a priori knowledge of the victim's telephone number. Position resolution was observed to within 1 km^2.

Bindschaedler et al. [80] use a group of 802.11 access ports to eavesdrop on proximate target smartphones in order to evaluate mixing zone effectiveness. Data collection includes device time, location, device identifier and content. Although victims may attempt to hide via a mix-zone network where MAC addresses are synchronously changing (assuming sufficient group membership), tracking can be achieved. This attack requires collusion of multiple APs and Wi-Fi or equivalent communications mechanisms. This may be impractical to set up exclusive of the most sophisticated attackers.

Hybrid Attacks: There are a number of works [81, 82, 83, 84, 85, 86, 87] that combine WLSS data with motion / inertial sensors to infer user location, mode of transit, orientation and behavior. Of those surveyed, we find best case accuracies near 80%. Although positional accuracy benefits offered by these mechanisms are interesting, these attacks generally require obtaining a 'fix' via WLSS functionality prior to leveraging sensor data. This exposes the attacker to WLSS discovery mechanisms.

Zhang et al. [88] developed the SensTrack system which identifies turning points using a smartphone's Accelerometer to determine speed, distance, and orientation. Additionally, they use sensors with adaptive Wi-Fi and GPS switching to address location contexts where GPS is less effective (i.e. indoor locations). Their system achieved prediction errors of nominally 3.128 meters versus 5 for good GPS signal strength. This approach assumes some location predetermination using GPS for initial reference position. Furthermore, the short distances within a building do not offer the challenges one realizes in the spatial-temporal context of driving a vehicle.

Sensor Only Attacks: The following attacks are most representative of our approach as they rely entirely on zero-permission sensor sources. Table 3.5 summarizes the scalability and inference accuracy of these related works in comparison to our attack.

Han et al. [21] suggested a method of location inference using the Accelerometer and Magnetometer. Leveraging a probabilistic dead reckoning method called Probabilistic Inertial Navigation (ProbIN), they mapped probability of displacement to probability of motion. Training data associates sensor data with map truth. Resolution is observed approaching 200 meters, the length of a typical city block. Their small sample size limited

Table 3.5: Summary of the scalability and inference accuracy of related works in comparison to our location tracking attack.

	Scalable to Cities?	**Inference Accuracy**
ProbIN [21]	Not scalable, needs sensor data mapped to different roads for training	Location resolution of 200 meters
Nawaz et al. [22]	Not scalable, uses DTW that requires significant computing power	Accurately clustered 43 routes in 8 clusters
Zhou et al. [89]	Not practical, (1) traffic variation impacts accuracy and (2) requires navigation app	Accuracy of 70% in a small sample set
PowerSpy [19]	Not scalable, needs power profile mapped to different locations for training	Accuracy of 66% with moderate count of apps, 20% with more apps
This Attack	Scalable, algorithm searches large routes in large cities in seconds	More than 50% of exact routes in top 10 search results for 8 out of 11 cities

the experimental path length range to between 1 km and 9.7 km. Although claiming better accuracy than achievable using Wi-Fi or cellular techniques, their approach greatly depends on acquiring training data which may present a resource challenge (i.e. time and labor) in large scale scenarios.

Nawaz et al. [22] demonstrate that a smartphone's Accelerometer and Gyroscope can be used to identify 'significant' journeys independent of phone orientation and traffic. This is because Gyroscope signatures obtained from multiple journeys of the same route exhibit similar patterns that differ only in amplitude and time compression or expansion. They apply Dynamic Time Warping (DTW) to calculate the distance between various journeys and use a k-medoids clustering approach to cluster similar routes together. A route is labeled as significant if it is traveled more times than a predefined threshold. They test this technique for two cities using 43 real driving experiments and showed that the routes were accurately clustered in 8 clusters defined for the two cities. This attack presents a resource challenge in large scale scenarios because DTW has a time complexity of $O(mn)$, where m and n are the number of Gyroscope samples in the two routes. For any small or large city, comparing two short 10 minute routes at a low Gyroscope sampling rate of 20 Hz (totaling $12,000$ samples) requires $144,000,000$ iterations. Our attack is

significantly more efficient as even large cities like Atlanta require a hundredth of the iterations in the worst case, i.e., $1,579,350$ iterations assuming 10 turns, 15 edges per vertex and no filtering. The use of filtering significantly decreases the iterations. Grid road networks are addressed differently as they depend on turn count as a uniqueness metric and suggest that their technique is effective for reasonably long routes because such routes exhibit a unique sequence of turns even when individual turns are similar.

Zhou et al. [89] describe a novel technique that analyzes verbal directions provided by a GPS based navigation app. Using a second zero-permissions app, they measure speaker on/off times controlled by the navigation app. The attacker can infer which course a driver took due to the duration of these audible driving instructions. Permission for speaker usage is not required as of this writing. Associating talk time to an off-board synthesized instruction driving set yields a 30% false positive rate over a small sample size (7 out of 10 correct). This approach requires the use of a voice enabled navigation system. Furthermore, it assumes that the navigation app is trustworthy.

Michalevsky et al. [19] introduce a power based scheme, called PowerSpy, that distinguishes a user route from a set of possible routes in real-time. Furthermore, they attempt to infer new routes by constructing projected route power profiles that are aggregated from shorter, known segment power profiles, all using 3G networks. With a 'modest' number of applications running, they achieve accurate results in 2/3 of the scenarios while the results degrade to an accuracy of 1/5 with additional active applications such as Facebook and Skype. In addition, they are limited by the need to provide data to the learning machine which itself limits scalability in obtaining training data.

Behavior Analysis: This research area involves determining user modality from smartphone sensors. For example, ergonomic / activity identification is discussed in [90]. The authors use learned data from walking, jogging, climbing stairs, sitting, and standing to ascertain user activity. They identified and collected data for 43 features from a 29 person sample set. Raw data was evaluated using the WEKA data mining tool suite to develop decision tree, logistic and regression and multilayer neural network models. Excluding motions associated with moving up and down stairs, the method can identify activity

nearly 90% of the time. Although of a single modality and reasonably well suited for human activity identification, this has limited ability to ascertain paths with much less start and stop points.

Lee and Mase [91] studied the feasibility of detecting user behavior such as sitting, standing, walking on level ground, going up or down a stairway as well as determining the number of steps taken to infer a person's location in an indoor environment. They developed a system using the Accelerometer and Gyroscope sensors to measure the forward and upward acceleration and angle of the user's legs. Additionally, the compass is used to determine the direction of movement. The phone is mounted on different body locations and a dead-reckoning method is applied to estimate the user's location. The authors show that their system efficiently calculated the number of steps and location for eight individuals, using a predefined database of selected locations in an office environment. They claim a high recognition ratio of 91.8% for ten unique location transitions.

Other Works of Interest: Two additional works are noteworthy. They include a pattern matching / machine vision approach to path traversal tracking and a framework to measure the effectiveness of the attack.

There are numerous examples in the literature for matching shapes, patterns and contours. We identify one here for this discussion. Kupeev et al. [92] decomposed shape contours in terms of segments for purposes of determining similarity of contours. They were able to analyze 24 shape distances with 32 unique quantized rotation angles against one another. The error rate appeared to be less than 10%. Of importance is the limited use of this technique observed in the location privacy space. This approach's weaknesses are similar to other contour matching solutions in that the subtle differences in road contours may not be distinguishable between similar yet geographically separate roads.

Shokri et al. [93] suggest a framework for scoring location privacy protection mechanisms. Here, they define a triad taxonomy of accuracy, certainty and correctness where the latter represents the metric that determines the privacy of user. To our knowledge, this is the first significant attempt at establishing an evaluation framework for comparing privacy schemes.

3.6 Conclusion

We modeled the problem of tracking vehicular users as the problem of identifying the most likely route on a graph derived from the city's roads public database. The performance results of our algorithms, both simulations and experimental, indicate that in most cities a significant number of users are vulnerable to tracking by seemingly innocuous applications that do not request permissions to any sensitive information. We believe that this calls for rigorous methods and tools to mitigate side-channel attacks making use of mobile phones sensors.

Chapter 4

Mitigating Location Leakage with Dynamic App Sandboxing

This chapter describes the design and implementation of the MATRIX framework for Android that addresses some current privacy protection weaknesses in the Android ecosystem, and provides users with a tool to analyze how apps access their private information as well as the capability to change how certain untrusted apps receive location and sensor data. The chapter is structured as follows. In Section 4.1, we discuss the status of current Android privacy and protection mechanisms with a special focus on location and sensor information. In Section 4.2, we outline the high-level design of MATRIX and discuss how it addresses weaknesses in current privacy protection mechanisms. In Section 4.3, we describe some of the previous related work. Section 4.4 provides a detailed description of MATRIX's architecture and services. Section 4.5 provides a detailed description of our technique for generating realistic privacy-preserving synthetic user identities and mobility trajectories. In Section 4.6, we outline the evaluation metrics and present the results of our evaluations. We conclude in Section 4.7.

4.1 Privacy in Android Location and Sensors

This section provides a background on Android location and sensor specific APIs and current privacy protection schemes in the context of MATRIX. The weaknesses in these schemes are also discussed.

4.1.1 Android Location & Sensor APIs

The MATRIX framework requires auditing location and sensor API calls and changing the location information reported to an app in some contexts. This information can be

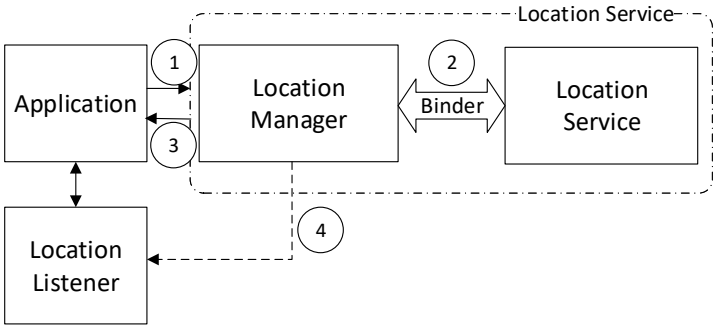


Figure 4.1: A high-level interaction of an app, the Location Manager and the Location Service in an Android device.

audited / modified on an Android device by monitoring specific APIs.

Location information can be requested using four set of APIs and their communication occurs as shown in Figure 4.1. The `LocationManager` is the core API that is provided by default in all versions of the Android SDK. The `FusedLocationProviderClient`, `FusedLocationProviderApi` (deprecated) and the `LocationClient` (deprecated) are provided by Google Play services as recommended closed source alternatives that consume less battery for higher accuracy data. All these APIs contain certain `request*` and `remove*` calls that allow apps to register and unregister for receiving location updates (e.g., `requestLocationUpdates` in `LocationManager`) ①. These managers communicate with the `LocationManagerService`, a privileged service that runs within the system context and communicates with the drivers over the `Binder` interface ②. The manager and the service check whether the app has the required permissions (`ACCESS_FINE_LOCATION` or `ACCESS_COARSE_LOCATION`) to request location updates and registers the app's listener. These listeners are implementation of the `LocationListener` interface, `PendingIntent` or `LocationCallback` classes in Android ③. Once registered, location information is sent asynchronously to the listeners based on the criteria set by the app (e.g., quality, rate, latency) ④. The managers also contain additional calls like `getLastLocation` that can return a location update immediately.

Sensor information (e.g., Accelerometer, Gyroscope, Magnetometer) can be requested

in much the same way as the location managers, by using the `SensorManager` API. The listener for sensor updates are an implementation of the `SensorEventListener` interface in Android. It is important to note that access to these sensors does not require any permissions in any versions of Android. Also, these sensors can be accessed by apps in the background, without any notification or visual cues to the user.

4.1.2 Location Privacy Protections

Android implements some location privacy protection mechanisms that are aimed to give users the capability to control how and whether certain apps can access their location data. These protections are discussed below.

Permissions Model: The most important location privacy protection is implemented in the form of permissions that apps must request in order to access location services. Android has two permissions that can be used to obtain the user's location information: `ACCESS_FINE_LOCATION` or `ACCESS_COARSE_LOCATION` or both. The former allows apps to access high accuracy location information, while the latter provides obfuscated location information to hide the user's location. Before Android `Marshmallow`, these location permissions were requested by the app at install time giving a user the option to either install or reject the app completely. Recently, Android switched to a run-time permissions model where the permissions are requested when an app accesses location specific APIs, giving the user the option to deny that specific call and continue using the app.

GPS Activity Notification: The Android operating system displays a notification icon on the notification bar of the device, whenever any app requests location information from the `GPS` location provider. This notification can also be expanded in some versions of Android to reveal the location data received, and the accuracy of this data.

Location Obfuscation: The Android operating system implements a location obfuscation scheme to hide a user's location from apps using just the `ACCESS_COARSE_LOCATION` permission. The implementation is in `com.android.server.location.LocationFudger` under the Android source tree [94]. We analyzed this code to find that the real location information is obfuscated in two steps. First, a random offset is applied to the location to

mitigate against accurate detection of grid transitions when a user crosses a grid boundary. This offset is changed slowly over time (e.g., once every hour) to mitigate against location inference attacks, where numerous samples are collected over time and averaged to get accurate location. Second, the primary means of obfuscation is to snap the offset data (already mitigated against grid transitions) to a grid. This grid radius chosen by most recent versions of Android is 2000m. We call this technique a *space-time* snapping.

4.1.3 Weaknesses in Current Protections

The privacy protection mechanisms discussed above are not sufficient for protecting a user's privacy. Some of their weaknesses are discussed below. Note that these weaknesses are labeled (**W#**) for ease of referring to them in the next sections. Moreover, App stores (e.g., Google Play Store) do not provide enough information about the privacy practices of an app, and entrust the decision of installing the app on the user. Without any information, users are more than likely to install an app if they require the services provided by that app.

Weak Permissions Model (W1): The dynamic permission model implemented by Android is a good step in notifying users about access to their location information. However, this protection is limited as users can set the option to always allow access. This means that the user will not be notified about location access again even if the app's context has changed, i.e., location is accessed from another activity or from a background service, or a previously benign app is updated with a privacy intrusive version.

Weak Location Activity Notification (W2): The location access notification icon is displayed only when an app registers for continuous location updates. Other means of location access can bypass this protection. One such example is the `getLastKnownLocation` call in `LocationManager` which can be invoked numerous times for receiving continuous location updates. Furthermore, the notification simply indicates that some app has access to location and no further information is given to the user to make privacy-aware decisions. Also, no versions of Android display notifications for sensor access using the `SensorManager` even when they are now known to leak location information.

Non-existent Auditing Capabilities (W3): Android does not provide a framework to audit how apps access a user's private information. This capability was added in Android `Jellybean (4.3)` in a permission manager called `App Ops`, but later removed from Android `KitKat` onwards. The removal might have been caused by a substantial number of apps crashing, when specific permissions were denied to those apps.

Restricted Privacy Preferences (W4): Android does not provide a framework for users to define their privacy preferences for apps installed on their device. Users with Android `Marshmallow` onwards have the capability to deny location access to certain apps by disallowing location permissions. In earlier versions of Android, users could simply deny location access completely on the device. While this provides privacy guarantees to users, certain apps can then deny service to the users. One may argue that users should simply not use apps that they don't trust. However, there are situations in which users do not wish to disclose their locations, in particular at some moments in time, and still require the app. One example of this is when the app is turned-off or in the background.

Location Granularity Settings (W5): Android implements two accuracy levels for location data which apps can request. For fine location, an app can receive location updates with a granularity of up to 3 meters. For coarse location, it can receive obfuscated location updates with a minimum accuracy of a city block. This obfuscated location still leaks some information about the user's location. There is currently no mechanism for users to completely hide their location by sending out synthetic information.

4.2 High-Level Approach

The MATRIX framework was implemented with the objective of addressing all location privacy protection weaknesses in modern Android operating system. To the best of our knowledge, our framework is currently the only one to provide real-time visual notifications of *location* and *sensor* access activity to the users. The notification bar is updated whenever an app accesses these sensors and displays which apps are actively accessing information from which sensors on the device **(W2)**. The framework implements fine auditing capabilities designed for both end-users and researchers / security apps desir-

ing to assess the privacy posture of installed apps on the device. End-users can view all location and sensor access information as intuitive graphs. Other apps can get access to the audit logs via a permission protected secure API **(W3)**.

The MATRIX framework also implements a location privacy preference module that enables users to set the type of location data an installed app can receive. This module currently implements three settings for location granularity: Block level, City level and Synthetic **(W4)**. MATRIX is the first, to the best of our knowledge, to automatically generate realistic randomized privacy-preserving synthetic identities and trajectories for users by modeling a user's mobility patterns as a finite state machine, modeling timing constraints as a Linear Program, generating state transitions as a graph route with historical traffic information, and incorporating user driving behavior in the routes **(W5)**.

MATRIX uses a different approach for location permissions. It relies on the default permission manager provided by Android `Marshmallow` and onwards, but restricts location access for background apps by default. Instead of completely denying location information, it detects if the requesting app is in the background and provides it the last location fix that the app received in foreground to prevent it from tracking users **(W1)**.

The high-level design of the framework comprises of four main modules: an API Call Interceptor Service, an App-activity PrivoScope Service, a Synthetic Location Service, and a Synthetic Location Provider. The API Call Interceptor Service intercepts calls to location and sensor APIs so that their parameters can be logged or modified. We implemented this service using the Xposed Framework [95] as it is open-source, well supported and popular among Android users. The App-activity PrivoScope Service generates and logs events when a privacy sensitive API is invoked. It provides a graphical interface to the users to monitor, visualize, audit, and analyze when and how an app accesses their location data and phone sensors. It also exposes a permission-protected API that allows other security apps installed on the smartphone to get real-time information about which apps access the location and sensors information. The Synthetic Location Service provides an interface to the user to set their location privacy preferences for any app. The last module, the Synthetic Location Provider generates realistic privacy preserving synthetic identities

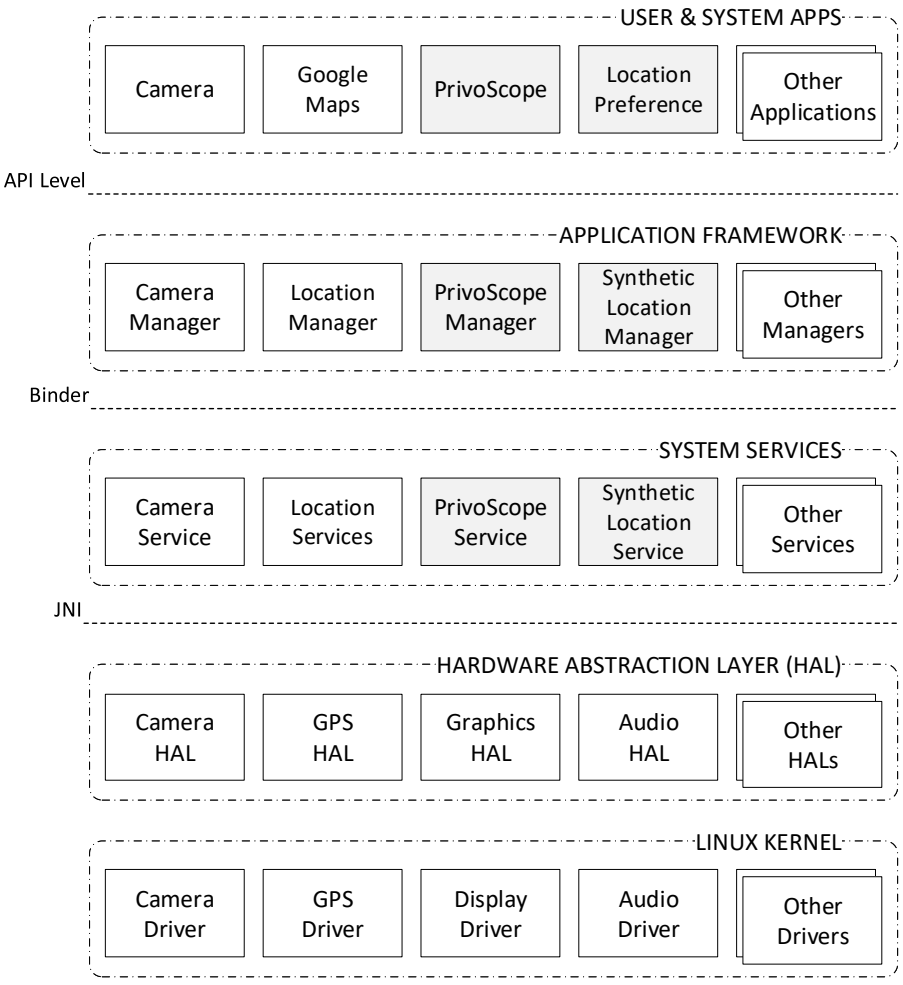


Figure 4.2: The MATRIX framework integration into the Android ecosystem.

and mobility trajectories for users and communicates with the Synthetic Location Service to provide it obfuscated / synthetic locations whenever the service requests for it.

Figure 4.2 shows how the MATRIX framework integrates into the Android ecosystem. The PrivoScope Service and Synthetic Location Service are implemented as system services that run in the same context as other system services like Camera and Location services. These services are registered in the system server registry and start whenever the device boots up. Apps installed on the smartphone interact with these services using public APIs exposed by the PrivoScope Manager and the Synthetic Location Manager. These managers are loaded inside the app's process and communicate with the corre-

sponding services using the `Binder` IPC. The services implement the protections that ensure that only authorized apps use their services. At the user level, MATRIX implements the PrivoScope GUI that provides a graphical interface to the users to analyze the app's privacy practices, and the Location Preference GUI that enables users to set their location privacy preferences. These activities also use the PrivoScope and Synthetic Location managers to communicate with the corresponding system services. The next sections describe some of the related works, the architecture of these services, and the synthetic identities and trajectories generation technique in detail.

4.3 Related Work

A large body of research has focused on mitigating location and other private information leakage attacks on Android devices. Most of these works are orthogonal to our system as their motivation and techniques differ. Examples of such work include, but are not limited to, recommending new security frameworks [2, 38, 39, 41, 44, 45], location obfuscation [23, 24, 25, 26, 27], location cloaking [28], generating dummy locations [29, 30, 31, 32, 33, 96], tainting sensitive data [46, 47], dynamic analysis [48, 49], static code analysis [50, 51, 52, 53], permissions analysis [54], application retrofitting [55, 56, 57], analyzing Internet traffic for sensitive information [58, 59], and even cryptographic techniques [60].

Synthesizing human mobility has also been studied in the context of opportunistic networks [97, 98, 99, 100], ad-hoc and vehicular wireless networks [101, 102, 103, 104], building group or community based mobility models [105, 106, 107, 108], predicting location of moving objects [109, 110], and for implementing efficient location update mechanisms [111, 112, 113, 114]. Some research has also focused on generating synthetic traces for user privacy [61, 63, 64], however, their traces are not very realistic as they do not satisfy the traffic constraints for different roads on different times of the day, nor take into account user driving patterns. In [63, 111], speed patterns from real GPS traces are simply superimposed on synthetic traces based on the street type without accounting for traffic conditions of the road. These speed patterns can also be repeated and can be detected. In [61], the synthetic traces are derived from real traces which does not apply in our context of generating completely synthetic identities for users. We describe some of the closest

works to our approach in more detail.

Beresford et al. [40] addressed weakness **W1** (cf. Section 4.1.3) on Android devices by implementing MockDroid, a modified version of Android 2.2.1 with a user controlled permissions manager. The system allowed users to define mock permissions for apps installed on the device. The location mock permission was implemented to block all location fixes from reaching the app simulating a lack of available location information. The authors ran the system on 27 apps and showed that most of the apps continued to function with reduced functionality. This system is similar to the current permissions manager of Android, however, does not mitigate any other weaknesses (W2-5).

Agarwal and Hall [42] addressed weakness **W4** on iOS devices by implementing ProtectMyPrivacy (PMP) as a mitigation system for jailbroken devices. The system intercepted calls that access user's private data and could substitute user-defined anonymized data in it's place, based on the user's decision. They also implemented a crowd-sourced app-privacy recommendation system using recommendations from their users. One limitation of this system is that the anonymized data is selected by the user at run-time, and therefore unrealistic and random. This can be easily detected by an app. They also do not address weaknesses **W2** and **W3** that can help users make more informed privacy decisions regarding apps.

Liu et al. [43] addressed weaknesses **W1** and **W3** on rooted Android devices by implementing a system called Personalized Privacy Assistant (PPA). This system is a modified `App Ops` permission manager that shows an app's recent requests and also how often an app requested that resource in the past 7 days. The system uses this information to generate daily privacy nudges to motivate users to interact and change their privacy settings. They also implemented a privacy settings recommender system modeled using a SVM classifier and trained by their initial user's privacy settings. During evaluation, they showed that 78.7% of the recommendations were accepted by another group of participants. One weakness of this system is that the usage information is limited to the count of requests and does not provide additional information (e.g., time and durations of requests, was the app in the background?) to help users make informed privacy decisions.

Table 4.1: Summary of the protections implemented by current privacy protection systems in comparison with MATRIX.

	MockDroid	**PMP**	**PPA**	**Zheng et al.**	**Fawaz & Shin**	**MATRIX**
System Implemented	✓	✓	✓		✓	✓
Addresses Weakness 1 (W1)	✓		✓			✓
Addresses Weakness 2 (W2)						✓
Addresses Weakness 3 (W3)			✓			✓
Addresses Weakness 4 (W4)		✓			✓	✓
Addresses Weakness 5 (W5)				✓	✓	✓

Zheng et al. [104] address weakness **W5** by proposing an agenda driven mobility model that considers a person's daily social activities for motion generation. They derive this agenda from the National Household Travel Survey (NHTS) information by the U.S. Department of Transportation. The first agenda and all subsequent activities are based on the NHTS activity distribution, and addresses are picked at random from many addresses for the corresponding activity. The start time of the first agenda determines the schedule for the entire day and each activity starts immediately after the mean dwell time + longest transition time from previous activity. The route between two activities assumes a longest possible time given by the Dijkstra's algorithm and does not change for different traffic patterns at different times of the day, nor incorporates any user driving behavior.

Fawaz and Shin [2] address weaknesses **W4** and **W5** on Android devices by implementing LP-Guardian, a privacy protection framework, modifying the Android source. The framework changes location granularity of installed apps based on the threat posed by the app and its location granularity requirements. It automatically coarsens the location to a city level if it identifies a request from an A&A library, the app is in the background, or the app is a weather app. It synthesizes the location for fitness apps but preserves features of the actual route such as the distance traveled. The framework supplies a synthetic location if it determines that it is not safe to release the location. For other apps, the system checks with the user if they are comfortable with the location release and coarsens the location if they decline. The weakness with this system is that, unless chosen very carefully, the synthetic traces generated for real traces will not snap to streets

(e.g., different street lengths and curvatures) and can be easily detected as synthetic.

Table 4.1 displays a summary of the protections implemented by related works in comparison to our system. MATRIX significantly improves on these systems by addressing their weaknesses and all the privacy protection weaknesses in Android.

4.4 MATRIX Architecture

This section describes the architecture of the services implemented by the MATRIX framework, namely the API Call Interceptor Service, the App-activity PrivoScope service, and the Synthetic Location service. The synthetic location generation technique used by the Synthetic Location service is described in Section 4.5.

4.4.1 MATRIX API Call Interceptor

Implementing an audit framework on Android requires privileged access for intercepting API calls. Previous mitigation frameworks, with the exception of Boxify [38], were implemented by either modifying the Android source code, using rooted devices, or using third party frameworks such as the Xposed Framework. The Xposed framework adds an extended `app_process` executable in the `/system/bin` folder of the device when installed. This extended `app_process` adds an additional `jar` file to the classpath and calls methods even before the `main` method of the `Zygote` process is called. This allows apps to hook system method calls that are otherwise inaccessible from an app's process.

The Xposed framework approach is advantageous because it gives MATRIX the capability to 'hook' non-native methods and modify their functionality. It is also supported for different versions of Android ensuring that the system is portable. Another advantage is that it does not require root to function on the device. We developed a simple tool that automates the installation of the Xposed framework through a custom recovery (e.g., TWRP [115]), with neither rooting the phone, nor user intervention. The framework and TWRP recovery are both open-source and consistently analyzed and updated by a large community of Android users, making them quite reliable and secure.

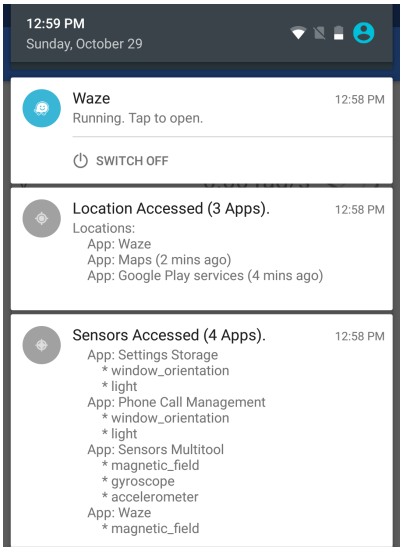
The MATRIX system uses the Xposed framework to intercept location and sensor

API calls. The framework exposes an abstract class called `XC_MethodHook` with two calls, `beforeHookedMethod` and `afterHookedMethod`, that can alter a method's execution. As the name implies, an implementation of `beforeHookedMethod` would execute before the actual method, while `afterHookedMethod` would execute after the actual method has executed. One example usage in our context is hooking the `requestLocationUpdates` method of `LocationManager` to generate an event in the `afterHookedMethod` call every time an app requests location updates. This event contains all the relevant information about the request and sent to the PrivoScope Service for logging and notification.

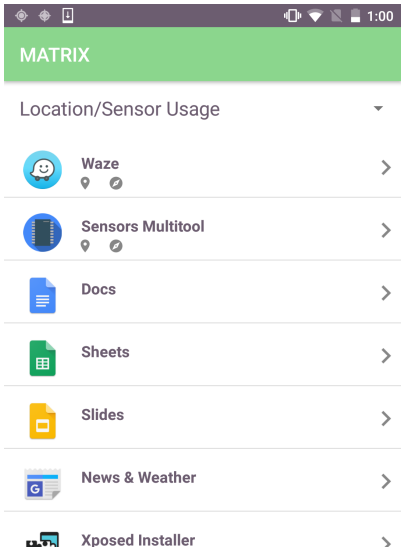
4.4.2 The App-activity PrivoScope Service

The PrivoScope service is implemented closely following the Android design paradigms. The goal is to implement an efficient system with minimal overhead that can easily integrate into the Android ecosystem. Another goal is to provide a modular audit framework that can be extended to incorporate other modules in the future. At a high level, this service uses the Xposed framework to intercept location and sensor APIs, generates events containing the audit details, adds the events to a database and displays real-time usage notifications. The service also exposes a permission protected API that other apps can register to get real-time and archived audit events. It also implements a GUI interface for the users to analyze app behavior on their device. The motivation is to help users make privacy aware decisions regarding installed apps. Figure 4.3 shows example screenshots of the PrivoScope GUI, where Figure 4.3a shows the PrivoScope real-time location and sensor usage notification, Figure 4.3b shows a list of installed apps sorted on most recent access of location and sensor APIs, Figure 4.3c shows an app's details, and Figure 4.3d shows a timeline of Accelerometer access by an app at different times. Note that the app's life-cycle is color coded for the user to differentiate between foreground and background accesses. Here, the blue color indicates that the app was in the foreground while gray would indicate background access. The evaluation and performance analysis of PrivoScope is reported in Section 4.6.2.

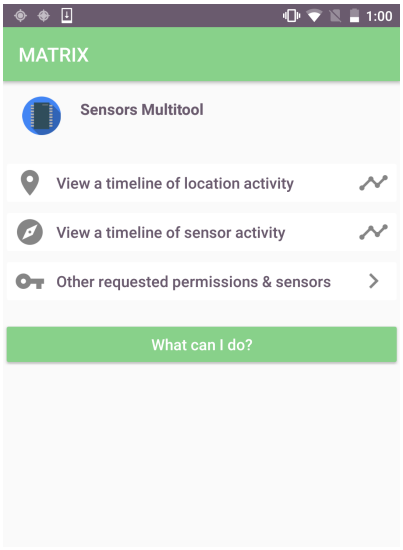
The architecture of the PrivoScope service for a `requestLocationUpdate` method call from `LocationManager` is shown in Figure 4.4. Note that the same flow applies to method



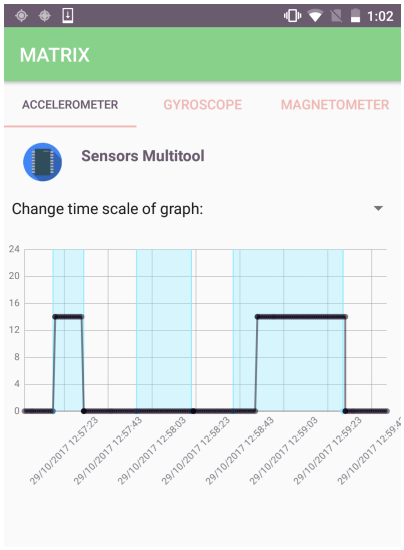
(a) PrivoScope Notification Bar



(b) App Selection Activity



(c) App Detail Activity



(d) Accelerometer Activity Timeline

Figure 4.3: Example screenshots of the MATRIX PrivoScope service's Graphical User Interface.

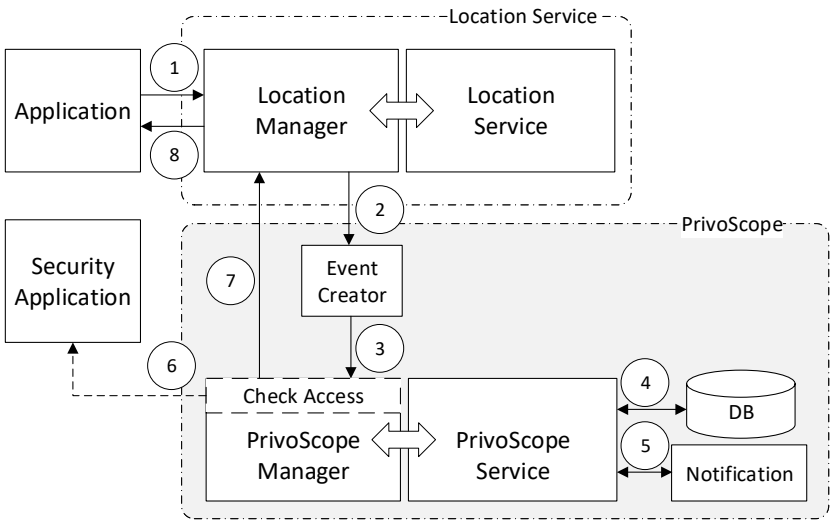


Figure 4.4: A high-level architecture diagram of the MATRIX PrivoScope service.

calls from the other location managers and the sensor manager. Like other Android services, the PrivoScope service implements a manager called `PrivoScopeManager` that exposes public APIs for other apps and a system service called `PrivoScopeService` that performs all the security sensitive operations and checks if apps have appropriate access rights for method calls.

The control flows like this: An app requests continuous location updates using the `requestLocationUpdate` method call from `LocationManager`. The manager and the privileged `LocationManagerService` validate the app's access by checking its requested permissions ①. Once access is validated, the API interceptor service generates an event containing all relevant information to be logged for auditing. All user privacy information contained by the request are ignored. For example, this specific event would contain *the system time, the app package name, the activity invoking the request, whether the app is background or foreground, the requested location provider, and the requested accuracy and sampling rate* ②. This event is then sent to the `PrivoScopeManager` for logging using an `addAuditEvent` method call exposed by the manager ③. The `PrivoScopeManager` forwards this event to the `PrivoScopeService` which validates whether the package name in the event is the same as the package name of the app making the request. This ensures

security as only apps generating an event can add the event. The event is discarded if the package names do not match and a `SecurityException` is thrown. In case of a successful match, the event is added to the service's database ④. The `PrivoScopeService` also sends this event to a Notification service that keeps track of all active apps accessing location and sensor APIs and updates the notification bar with this new event information ⑤. The `PrivoScopeManager` exposes a `requestAuditEvents` method call that other apps on the device can register for receiving real-time audit events. This call is protected using a custom permission called `GET_AUDIT_EVENTS` and apps must request this permission for access. The `PrivoScopeManager` sends the event to all registered apps that receive this event asynchronously using a `AuditEventListener` callback interface ⑥. Based on whether this event was successfully added to the database or not, the `addAuditEvent` method call returns a boolean value to the `LocationManager` ⑦. Note that steps ③ to ⑦ execute in a new thread to ensure that the app functionality and the performance is not impacted by PrivoScope. After step ③, the `requestLocationUpdate` method call simply terminates as its return type is a `void`. The other method calls and managers return the expected values and their functionality is not updated by PrivoScope ⑧.

4.4.3 The Synthetic Location Service

The architecture of the Synthetic Location service is shown in Figure 4.5, again in the context of receiving location updates from the `LocationManager` API. Note that the same flow applies to all the other location managers. The Synthetic Location service implements a manager called `SyntheticLocationManager` that exposes public APIs for other apps and a system service called `SyntheticLocationService` that manages and protects the database storing the user location preferences, and connects with the `LocationProvider` to request obfuscated / synthetic locations.

The control flows like this: When an app requests continuous location updates (with the correct permissions) using the `requestLocationUpdates` call from `LocationManager`, the first steps that occur are the listener registration (cf. Section 4.1.1) and addition of the audit event to the PrivoScope service's database (cf. Section 4.4.2). ①, ②. After registration is completed, all the location fixes generated by the `LocationManagerService`

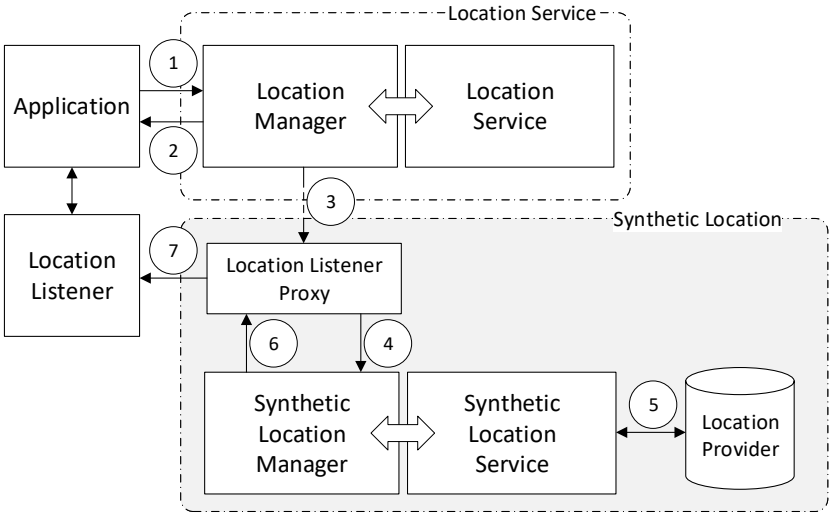
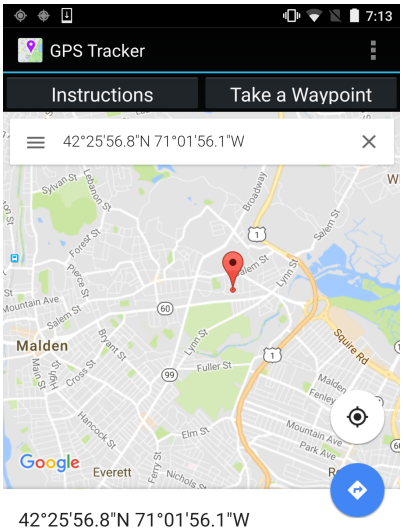
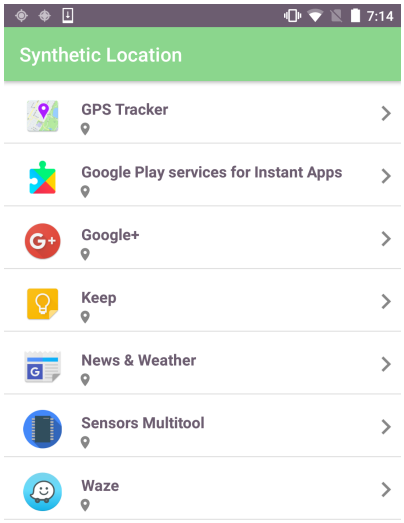


Figure 4.5: A high-level architecture diagram of the MATRIX Synthetic Location service.

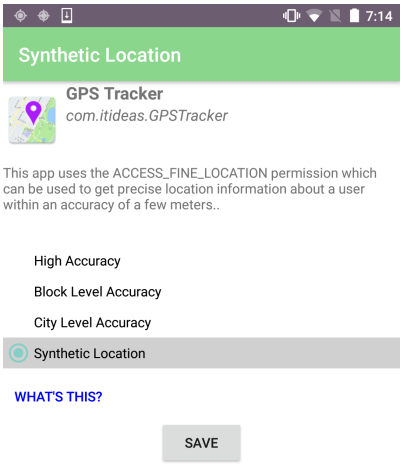
are typically sent asynchronously to the app's `LocationListener`, `PendingIntent` or `LocationCallback` implementation. In MATRIX, these location fixes are intercepted by a `LocationListenerProxy` that proxies it to the app's listener. The proxy works by hooking the `Location` object that is used by all the managers to send location fixes to the app's listener. This enables it to modify the location object before the app loads the information using the get* method calls (e.g., `getLatitude() and getLongitude()`) ③. The `LocationListenerProxy` requests the `SyntheticLocationManager` to provide an updated location for the app, based on the app's location preference set by the user. The manager forwards this request to the `SyntheticLocationService` that maintains and protects the database storing the user location preference for each app ④. The `SyntheticLocationService` looks up the user's location preferences in the database, and communicates with the `LocationProvider` to request an obfuscated / synthetic location if the user has chosen to receive such location information for the app. The default preference set for an app requesting fine location is block level obfuscated data (500m) ⑤. An updated location object is returned to the `SyntheticLocationService` which forwards it to the `SyntheticLocationManager`. The `SyntheticLocationManager` sends this location to the `LocationListenerProxy` that updates it before the user accesses the location ⑥, ⑦.



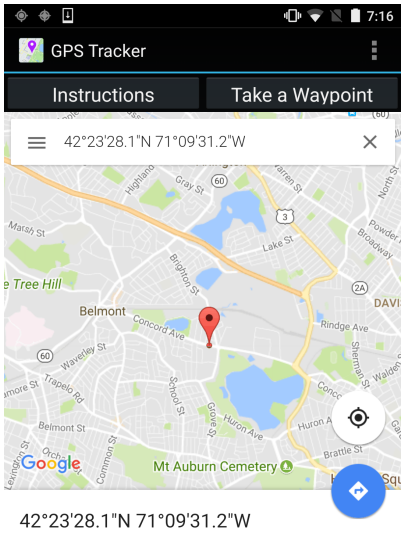
(a) Real Location



(b) App Selection Activity



(c) Location Preference



(d) Synthetic Location

Figure 4.6: Example screenshots of the MATRIX Synthetic Location service's Graphical User Interface.

The Synthetic Location service currently provides four settings for per-app location privacy: High Accuracy, Block Level Accuracy, City Level Accuracy and Synthetic Locations. The high accuracy option set for an app tells the service to not obfuscate or synthesize locations for this app. For block level and city level accuracy, we extended the default Android `LocationFudger` implementation to support different grid resolutions. We found this technique to be effective against location inference attacks. The current grid radius settings for block level and city level accuracy are $500m$ and $5000m$, respectively. Note that the High Accuracy and Block Level Accuracy options are only available for apps requesting fine location using the `ACCESS_FINE_LOCATION` permission. This is because apps that use `ACCESS_COARSE_LOCATION` permissions already receive coarser location data than that provided by the two options.

Figure 4.6 shows screenshots, illustrating the Synthetic Location service for a GPS tracking app. Note that this app is used for demonstrating how the service works because it displays the user location on the screen, and it is not a malicious app. Figure 4.6a shows the test app displaying the user's real location, Figure 4.6b shows the list of installed apps that request location permissions, Figure 4.6c shows the location privacy preference for the test app being changed to synthetic, and Figure 4.6d shows the test app now displaying a synthetic location in another city. This synthetic location is provided based on the time of the day and a realistic GPS trace created for the user for that specific day.

4.5 Generating Synthetic Trajectories

This section provides a detailed description of our technique for generating realistic privacy-preserving synthetic identities and mobility trajectories.

4.5.1 Modeling User States

A user's synthetic movements are defined as an automated probabilistic state machine with a finite set of S states $Q = \{Q_0, \ldots, Q_{S-1}\}$. The states, in this context, represent a sequence of tuples $\{(\text{Loc}(Q_i), t_{min,i}, a_{min,i}, a_{max,i})\}$, where $\text{Loc}(Q_i)$ is the geographic coordinates of state Q_i, $t_{min,i}$ is the minimum time spent in the state, and $a_{min,i}$, $a_{max,i}$ are the lower and upper time bounds for arrival at the state. The geographic coordinates of

the states are obtained from OpenStreetMap by parsing the 'building' and 'amenity' tags [116, 117] of all ways and nodes for the given area. For instance, a 'Home' state can be chosen as a way or node in OpenStreetMap whose building type is one of the following: 'apartments', 'house', 'residential', or 'bungalow'. Similarly, a 'Work' state can be chosen from the 'commercial' or 'industrial' tags. The other attributes are used for scheduling the user's activity for each day and set based on typical times that these activities occur. Note that the attributes are set to default values when they are unimportant for a state, i.e., $t_{min,i} = 0$, $a_{min,i} =$ 00:00:00, and $a_{max,i} =$ 23:59:59. In the simplest form, a state machine may contain just two synthetic states $Q = \{Q_0, Q_1\}$, where Q_0 = 'Home' and Q_1 = 'Work'. We label these as *significant* states as the user spends most of their time in one of these states. The geographic coordinates $\text{Loc}(Q_0)$ and $\text{Loc}(Q_1)$ are randomly chosen from the list of all locations with the relevant tags. Assuming no 'Work from Home' scenarios, the probabilities $P(Q_0)$ and $P(Q_1)$ of occurrence of these states is taken to be 1.

The state machine is made more realistic by adding synthetic states like Q_2 = 'School', Q_3 = 'Gas Station', Q_4 = 'Lunch' and Q_5 = 'Dinner'. We label these as *transitional* states because a user will temporarily visit these states when transitioning between *significant* states (i.e., Q_0 and Q_1). For a transitional state Q_i, the geographic coordinates $\text{Loc}(Q_i)$ is selected from a set of locations $\text{Loc} = \{Loc_1, \ldots, Loc_N\}$ with the relevant tags, such that its distance is shortest from the significant states, i.e., $\text{Loc}(Q_i) = \arg\min_{L \in \text{Loc}} d(L, \text{Loc}(Q_0)) + d(L, \text{Loc}(Q_1))$. Note that, unlike *significant* states, visits to *transitional* states are occasional based on some specific frequency of occurrence. This frequency, denoted by f_i, is derived from a uniform distribution $\mathcal{U}(l, u)$ with l and u as the bounds for the frequency of visits to that state (e.g., once a week to once a month). In case of 'Gas Station' specifically, the system chooses a random mileage m and gas capacity c, and calculates the frequency as the number of days a user can travel between the *significant* states before the gas level goes below $1/4^{th}$ of capacity, i.e., $f_3 = int(\frac{0.75mc}{d(\text{Loc}(Q_0),\text{Loc}(Q_1)) + d(\text{Loc}(Q_1),\text{Loc}(Q_0))})$. Assuming W workdays in a year, the probability of occurrence for any *transitional* state Q_i is then calculated as $P(Q_i) = (W/f_i)/W$.

The transition probability between states Q_i and Q_j, denoted by $\chi_{i,j}$, is equivalent to the compound probability of the two independent states, i.e., $P(\chi_{i,j}) = P(Q_i)P(Q_j)$.

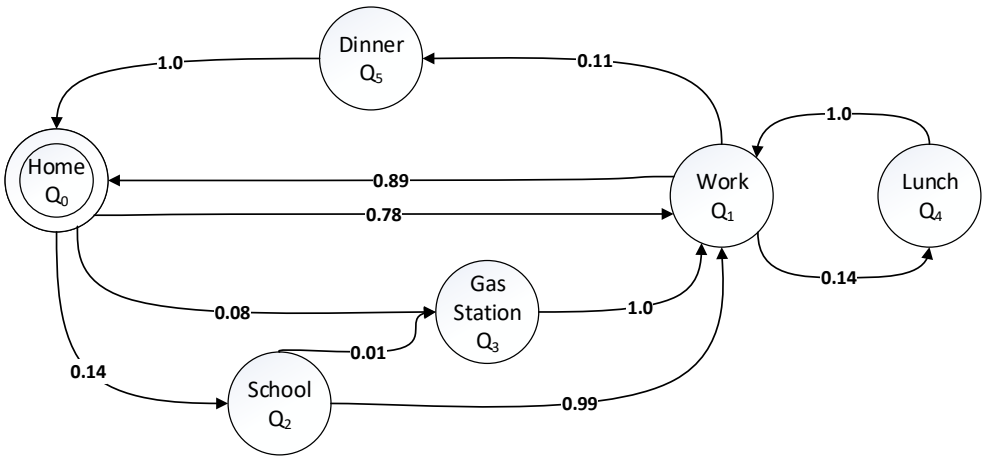


Figure 4.7: Example of a simplified finite state machine simulating a user's movements based on some transition probabilities.

The following conditions determine if a state Q_i can transition to state Q_j: (1) Q_i is a *significant* state and the originating state for Q_j, (2) Q_j is a *significant* state and the destination state for Q_i, or (3) the two states originate from the same *significant* state Q_s and distance $d(\text{Loc}(Q_s), \text{Loc}(Q_i)) < d(\text{Loc}(Q_s), \text{Loc}(Q_j))$. The *significant* states are always connected and their probabilities are calculated as $P(\chi_{0,1}) = 1 - \sum_{i=2}^{S-1} P(\chi_{0,i})$ and $P(\chi_{1,0}) = 1 - \sum_{i=2}^{S-1} P(\chi_{1,i})$, respectively. All other transitions have a probability of 0.

Note that users can go for 'Lunch' in the afternoon and 'Dinner' in the evening from the 'Work' state. If we use the same 'Work' state for both transitions, the probabilities are split when they clearly are different transitions. To address this, the 'Work' state is internally represented as two states: Q_{1a} for afternoon and Q_{1e} for evening. Also note that the model described here is for weekdays, and a similar model is created for weekends with a different set of states (e.g., the user may leave from 'Home' to watch a 'Movie', eat 'Dinner' and return 'Home').

Figure 4.7 provides an intuition for our automated finite state machine model. This specific model comprises of 6 states $Q = \{Q_0, \cdots, Q_5\}$ and their transition probabilities are shown. We see that it is possible to transition from state Q_0 to states Q_1, Q_2 or Q_3.

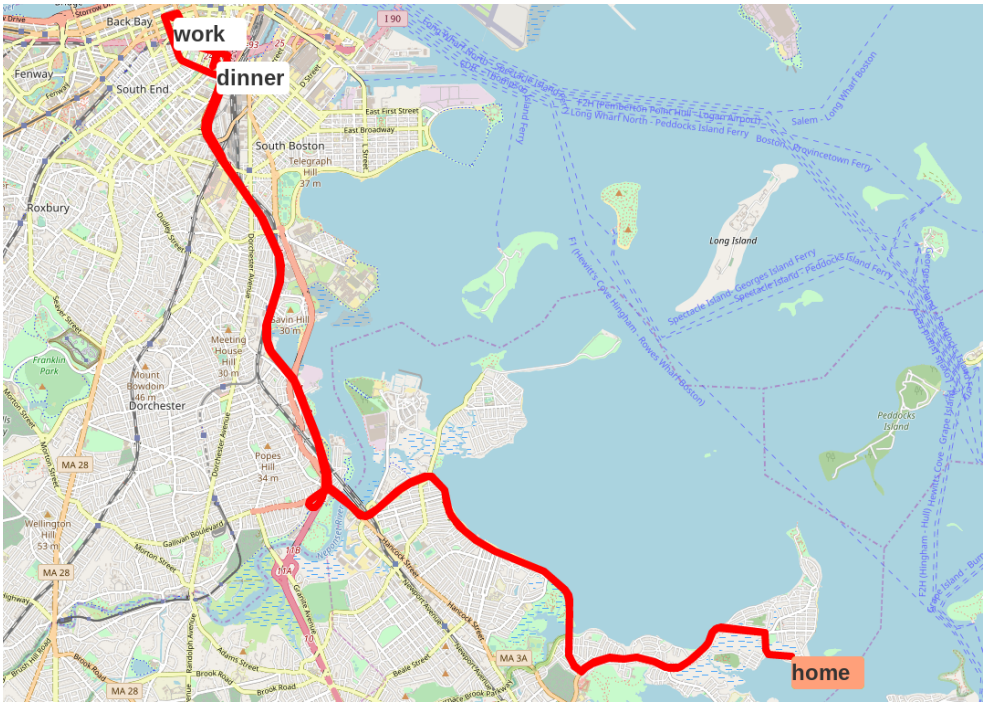
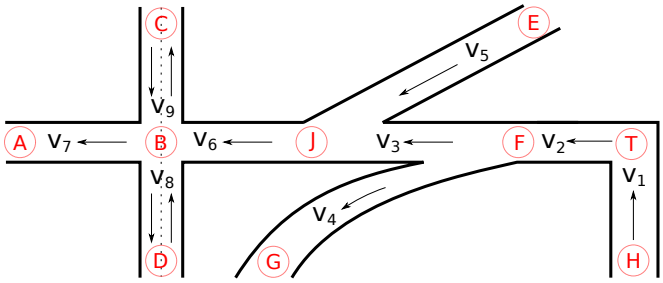


Figure 4.8: Example of a GPS trajectory generated for an entire day, given the state machine and transition probabilities in Figure 4.7.

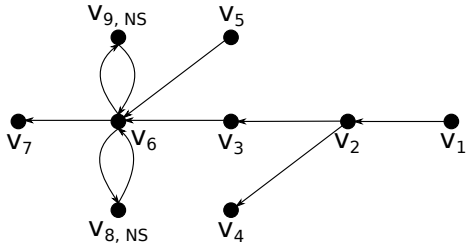
As the transition probability $P(\chi_{0,1})$ is 0.78, the model should typically choose state Q_1 $\approx$ 8 times out of 10. This makes sense as a user will mostly go to 'Work' from 'Home' but may sometimes need to drop their kids to 'School' or fill up gas at a 'Gas Station'. Figure 4.8 shows an example of a synthetic trajectory generated for a user for a day. On this particular day, the user goes from 'Home' to 'Work' in the morning and stops to eat 'Dinner' from 'Work' to 'Home' in the evening.

4.5.2 Graph Construction

The synthetic trajectories for a geographic area $\mathcal{G}$ is generated using a directed graph $G_{\mathcal{G}} = (V, E)$, constructed from the OpenStreetMap road network of that area. Recall that this geographic area can be represented as $\mathcal{G} = (\mathcal{B}, \mathcal{C}, \theta, \vartheta)$, where $\mathcal{B}$ is a set of atomic parts, and $\mathcal{C} = \{\chi = (r, r') | r, r' \in \mathcal{B}\}$ consists of ordered pair of connections $\chi = (r, r')$ indicating the connection between two atomic parts r and r'. The turn angle associated with a connection χ is given by function θ and the atomic part's curvature is given by $\vartheta(r)$



(a) Example Road Network.



(b) The corresponding Graph representation.

Figure 4.9: Example of a road network and its graph representation. The sections of road between intersections represent vertices v, and the intersections represent edges e.

(cf. Section 3.2). In this graph construction, we represent each atomic part $\vec{r}$ by a vertex $v \in V$ and each connection χ by an edge $e \in E$. A default speed limit is assigned to each atomic part $\vec{r}$ based on its 'highway' type in OpenStreetMap. For example, a 'motorway' highway symbolizes Interstates in USA which have speed limits $\approx 65mph$. The length, speed limit, and geographic coordinates of the atomic part $\vec{r}$ are stored as attributes of the corresponding vertex v. The length and speed limit are used to calculate the fastest time of travel between the endpoints. *Note that this is a one time initialization step for the area.*

Figure 4.9 shows an example road network and the corresponding graph construction. The graph is used to generate routes between two synthetic states using the *Dijkstra's* algorithm. One may argue that *Google Maps Directions API* can be used directly to generate routes instead of our graph, however, using the graph provides an advantage that these routes can be easily randomized by simply choosing a different edge of a vertex. It is also easier to specify additional waypoints to get more granular historical traffic from the Google API than what it currently provides.

4.5.3 Synthesizing the Trajectory

The finite state machine generated for a user and the graph constructed for an area are used for synthesizing mobility trajectories for the user. This is a 3 step process: (1) synthesize the user states for the entire day, (2) synthesize the schedule to satisfy the time constraints, and (3) synthesize the trajectory based on the schedule.

Synthesizing the user states: The state machine of a user is loaded every day to generate a route of the states the user will visit that day. This route always starts and ends at the initial state Q_0 ('Home') and traverses through Q_1 ('Work'), i.e., $R = [Q_0, \ldots, Q_1, \ldots, Q_0]$. The first state Q_0 can transition to any connected state Q_i based on the transition probabilities of Q_0. The state Q_i can then transition to any of its connected state Q_j based on the transition probabilities of Q_i, and so forth forming a chain that ends at the final state Q_0. Note that the construction technique of the state machine ensures that this route traverses through Q_1. Let $P(\chi_i) = \{P(\chi_{i,0}), \ldots, P(\chi_{i,S-1})\}$ denote the set of all transitional probabilities of state Q_i. To obtain the next state, the system first derives a random transitional probability from a uniform distribution $\mathcal{P} = \mathcal{U}(0,1)$. This probability $\mathcal{P}$ is then compared with the cumulative probabilities of all transitions in $P(\chi_i)$. A state Q_j is selected if $\mathcal{P}$ lies between the previous state's cumulative probability and its cumulative probability, i.e., $P(X \leq \chi_{i,j-1}) < \mathcal{P} \leq P(X \leq \chi_{i,j})$.

Synthesizing the schedule: A realistic schedule should satisfy the time constraints set for every state in a user's state machine, such as arriving at work between $8am$ and $9am$ or dropping children to school before $8{:}30am$. The schedule should also satisfy the amount of time spent in each state, such as working for at least $8hrs$. The schedule should also account for the time spent in transitioning from one state to the next, such as driving for $0.5hrs$ to get from home to work. All these constraints can be formulated as linear equalities or inequalities, therefore, defining the problem of scheduling as a Linear Program (LP). Let t_i^a and t_i^d be the arrival and departure times at / from state Q_i. The above constraints can be formulated as follows: arriving at state Q_i between $8am$ and $9am$ is formulated as $8am < t_i^a \leq 9am$, specifying that the user works at least $8hrs$ is formulated as $t_{i+1}^d - t_i^a \geq 8.0$, and the time spent in transitioning from home to work is

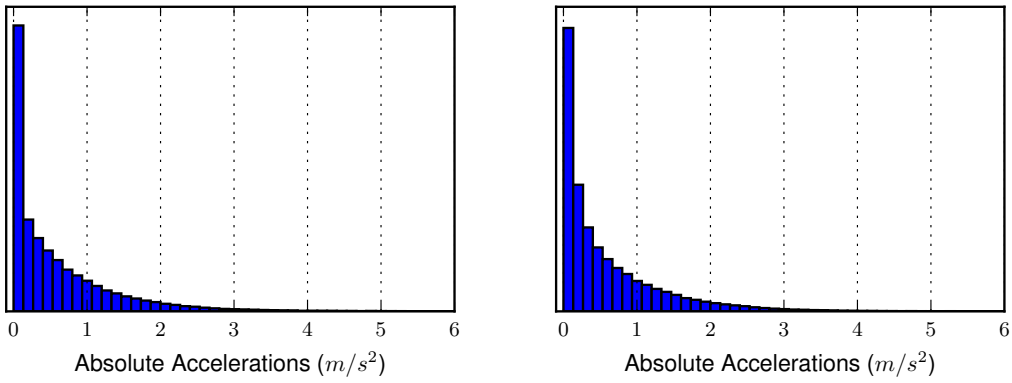
formulated as $t^a_{i+1} - t^d_i = 0.5$. Naturally, all the times are specified in UTC for consistency and bounded by the day's limits (i.e., 00:00:00 - 23:59:59).

This set of linear equality and inequality constraints define a *convex polytope* of all the schedules satisfying the state constraints, and the transition time constraints between the states. Let $T = (t^a_1, t^d_1, \ldots, t^a_S, t^d_S)$ denote a vector of all the arrival and departure time instants for a route containing S states. One simple way of finding a point on this polytope is by defining an objective function for the vector T with random coefficients, i.e., $c = (c_1, \ldots, c_S)$ where $c_i \in [-1, 1]$. Let $t(\chi_{i,j})$ denote the total time spent in transitioning between two states Q_i and Q_j. Also, recall that $t_{min,i}$ specifies the minimum time spent in state Q_i and $a_{min,i}$, $a_{max,i}$ specify the time bounds of arrival at the state. Using the above attributes, the LP is formally defined as:

$$
\begin{aligned}
\text{Maximize} \quad & \sum_{i=1}^{S} (c_i t^a_i + c_i t^d_i) && \text{where } c_i \in [-1, 1] \\
\text{Subject to:} \quad & a_{min,j} < t^a_j \leq a_{max,j} && \text{for } j = 1, 2, \ldots, S \\
& t^d_{j+1} - t^a_j \geq t_{min,j} && \text{for } j = 1, 2, \ldots, S-1 \\
& t^a_{j+1} - t^d_j = t(\chi_{j,j+1}) && \text{for } j = 1, 2, \ldots, S-1
\end{aligned}
$$

Solving this LP identifies a corner of the polytope but not a random element within it. If the coefficients of the objective function were repeated, the LP will output the same schedule. To address this, we compute a random point within the polytope by finding different corners of the polytope using random coefficients, and then computing a random linear combination of these corners. More precisely, let $C = \{C_1, \ldots, C_N\}$ denote a set of N corners of the polytope obtained using random coefficients, and let $r = \{r_1, \ldots, r_N\}$ denote a set of positive random numbers such that $\sum_{i=1}^{N} r_i = 1$. The random solution defining the user's schedule for that day is then calculated as $Sol = \sum_{i=1}^{N} r_i C_i$.

Note that as synthesizing the schedule using LP requires pre-calculated transition times $t(\chi_{i,j})$, the system calculates this time using the 'pessimistic' traffic model of *Google Maps Directions API*. The departure time is chosen as the mean of the time constraints for the start state. This typically gives us a worst case transition time between two states and can be used for scheduling. Note that for synthesizing the final trajectory, the 'best_guess'



(a) Real Accelerations

(b) Synthetic Accelerations

Figure 4.10: Distribution of the absolute values of accelerations for both Real ($\mu = 0.61$, $M = 0.34$, $\sigma = 0.79$) and Synthetic ($\mu = 0.61$, $M = 0.32$, $\sigma = 0.78$) routes.

traffic model is used which provides more accurate traffic representation.

Synthesizing the route between two states: The route between two synthetic states is generated using the graph $G_{\mathcal{G}} = (V, E)$ constructed for the area $\mathcal{G}$. The system uses the *Dijkstra's* algorithm to find the fastest route between the states, using the length and speed limit information present in each vertex v. The resulting route is split into multiple waypoints based on turns and stop signs (extracted from OpenStreetMap). These waypoints are given as input to the *Google Maps Directions API* to obtain historical traffic information about the route. The departure time is specified based on the schedule generated for that day. The route obtained from the Google API consists of multiple steps and can be represented as $R = [r_1, \ldots, r_S]$, where S denotes the number of steps. Each step r_i is attributed with geographic and traffic related information $r_i = (\mathcal{B}, d_{step}, t_{step})_i$, where $\mathcal{B}$ is the list of geographic coordinates of this step, d_{step} is the length of this step, and t_{step} is the time to traverse this step.

To generate realistic trajectories, all steps of a route must incorporate user driving behavior while also adhering to the step's traffic constraints, i.e., d_{step} and t_{step}. To understand the user driving behavior, we analyzed 400 driving routes collected from 2 drivers and 4 phones (LG Nexus 5, LG Nexus 5X, Samsung Note 4, and Google Pixel). These routes covered a distance of $\approx 1400kms$ in and around the city of Boston, consisting of

both highway and internal roads, as well as peak and off-peak hours. The acceleration and speed information were extracted from these routes for every second to analyze their distribution. We found the speeds to be randomly distributed, however, the absolute values of accelerations approximate to an exponential distribution (mean $\mu = 0.61$, median $M = 0.34$, and standard deviation $\sigma = 0.79$) as shown in Figure 4.10a. Note that the distribution is an approximation and not truly exponential because $\mu < \sigma$, where $\mu = \sigma$ is a property of exponential distributions. Analyzing individual routes, the range of means of the absolute accelerations, denoted by $[|\bar{a}|_{min}, |\bar{a}|_{max}]$, varied between $0.1m/s^2$ and $1.1m/s^2$. The range of standard deviations of the absolute accelerations, denoted by $[\sigma(|a|)_{min}, \sigma(|a|)_{max}]$, were between $0.4m/s^2$ and $1.1m/s^2$. The bounds of all acceleration values, denoted by $[a_{min}, a_{max}]$, were between $-7m/s^2$ and $7m/s^2$. Also, the means of the acceleration values were $\approx 0m/s^2$ for every individual route.

The above constraints can be formulated as a list of equalities and inequalities, this time defining a non-linear constraint optimization problem. Such problems can be solved by using Sequential Quadratic Programming (SQP) methods. Let $a = (a_1, \ldots, a_N)$ denote a vector of acceleration values for each step, where N denotes the travel time of the step, i.e., $N = int(t_{step})$. Let v_0 denote the initial speed coming into this step and $v = (v_1, \ldots, v_N)$ denote a vector of speeds calculated from v_0 and the vector a. The objective of this optimization is to find an optimal vector a that minimizes $|\bar{v} - (d_{step}/t_{step})| < \Delta$ to adhere to the traffic constraints, where $\bar{v}$ is the mean of vector v, and d_{step}, t_{step} represent the step's distance and time. The Δ is a threshold that determines whether the minimized objective function value is acceptable. All rejected optimizations are retried with a higher number of iterations till a valid solution satisfying the threshold is found. We observed that this optimization typically yields an optimal vector a that approaches the lower mean bound of the absolute accelerations $|\bar{a}|_{min}$, for most optimizations. To address this, we derive a new lower mean bound for every route from a uniform distribution and use the following range for optimization: $[|\bar{a}|_{rand}, |\bar{a}|_{max}]$, where $|\bar{a}|_{rand} = \mathcal{U}(|\bar{a}|_{min}, |\bar{a}|_{max} - \delta)$, and δ is a small constant to ensure that $|\bar{a}|_{rand} < |\bar{a}|_{max}$. The optimal vectors a_i for every step i are merged to represent the route's accelerations. Note that a bounded constraint of the form $x_1 \leq x \leq x_2$ can be rewritten as $(x_2 - x)(x - x_1) \geq 0$ for simplifying the constraint

for the solver. Using the above attributes, the route optimization for each step is formally defined as:

$$
\begin{aligned}
\text{Minimize} \quad & |\bar{v} - (d_{step}/t_{step})| \\
\text{Subject to:} \quad & \bar{a} = 0 \\
& (|\bar{a}|_{max} - |\bar{a}|)(|\bar{a}| - |\bar{a}|_{rand}) \geq 0 \\
& (\sigma(|a|)_{max} - \sigma(|a|))(\sigma(|a|) - \sigma(|a|)_{min}) \geq 0 \\
& \sigma(|a|) - |\bar{a}| \geq 0 \\
\text{Bounds:} \quad & a_{min} \leq a_j \leq a_{max} \qquad \text{for } j = 1, 2, \ldots, N
\end{aligned}
$$

Some additional constraints applied to the optimization are that $v_0 = 0$ for the first step and $v_N = 0$ for the last step of the route. This optimization is improved by providing an initial guess of bounded accelerations derived from a gaussian distribution $\mathcal{N}(\bar{v}', 2)$, where $\mu = \bar{v}'$ is the mean step speed, i.e., $\bar{v}' = d_{step}/t_{step}$, and $\sigma = 2m/s$ is the standard deviation of the speed. Figure 4.10b shows the distribution of the absolute accelerations generated for the synthetic trajectories. We can observe that the parameters and shape of the distribution closely follows the parameters and shape of the real distribution.

Note that this work uses a linear model for synthesizing walks from the state's geographic coordinates to a vertex on the graph, and vice versa. The vertex containing a point nearest to the state's coordinates is chosen, and the driving route is started or stopped at this point. This simple model assumes a constant walking speed as our main focus was on driving. We plan to study models for generating realistic walk patterns in the future. Also note that as GPS data accuracy varies, a small random gaussian noise is added to each coordinate of the final trajectory.

4.6 Evaluation

In this section, we evaluate the MATRIX framework using the following metrics: the portability and stability of the framework for 1000 popular apps, the performance overheads of the framework, detection of synthetic trajectories by regular users, detection of synthetic trajectories by 10 popular location based apps, and the detection of synthetic

Table 4.2: Results of the Stability test for the MATRIX framework using 1000 popular Android apps on 4 smartphones.

Phone	Version	No Install	Success	Failure
HTC One M7	Lollipop	0	892	108
		0	894	106
HTC One M9	Marshmallow	15	796	189
		15	791	194
LG Nexus 5	Lollipop	0	938	62
		0	944	56
LG Nexus 5X	Marshmallow	0	851	149
		0	848	152

trajectories by Machine Learning algorithms. The first two metrics validate the stability and performance of the framework, while the rest validate the realism of the synthetic trajectories generated by the framework.

4.6.1 Framework Portability and Stability

The MATRIX framework is compatible with Android `KitKat` (i.e., SDK level 19) and onwards. It has been tested to work with Xposed Framework API versions 82 to 88 (current). These Xposed versions are also compatible with Android `KitKat` and onwards. This implies that the MATRIX framework can be ported to $\approx$ 93% of all Android devices globally [118], without any modifications[1].

The framework stability was evaluated using 4 smartphones: a HTC One M7 running `Lollipop`, a HTC One M9 running `Marshmallow`, a LG Nexus 5 running `Lollipop`, and a LG Nexus 5X running `Marshmallow`. The test was performed on 1000 randomly chosen popular apps from the Google Play Store. All these apps had a minimum rating of 4.0 and a minimum vote count of 10,000 users. From these apps, 583 requested fine or coarse location permissions and the remaining accessed sensor data using the `SensorManager` API. These 1000 apps were first run successively on a stock Android version of these smart-

[1]According to the Android Dashboard (as of December 10, 2017), about 93.2% of Android devices run `KitKat` or above.

phones, using an automated UI application exerciser tool called Android Monkey [119]. Then, the 1000 apps were re-run on the same phones with MATRIX installed (including Xposed) to monitor how many additional apps crash or fail to execute. The monkey tool was configured with the same settings for both these tests ($seed = 1$, $num\_events = 2500$) to ensure that the same set of pseudo-random events were generated.

Table 4.2 shows the results of the stability test for all the smartphones. The first row for each phone shows the test results for the stock version and the second row shows the test results for MATRIX. All the apps installed and ran on every phone except for 15 apps on the HTC One M9 (possibly due to compatibility reasons). The number of successful monkey runs are very similar in both the tests with the stock version performing better on two phones and the MATRIX version performing better on the other two. We analyzed the errors / crashes manually to check for Xposed or MATRIX specific errors and did not find any. This validates that MATRIX remains stable and runs as expected for different apps and under heavy usage.

4.6.2 Framework Performance

The MATRIX framework was extensively tested for performance overheads occurring from the most expensive operations of the system. We identified 3 potential performance bottlenecks in our system: (1) the API interception function using the Xposed framework; (2) the add audit event function of the PrivoScope service; and (3) the location provider function of the Synthetic Location service. We implemented a test app that invoked these functions 1 million times to test performance. The execution time was calculated as the difference between two `System.nanoTime` method calls placed immediately before and after the function execution. The API interception bottleneck is essentially caused by the Xposed framework loading and hooking method calls. To evaluate its performance, we created an empty method inside our system and hooked it using the Xposed framework.

Table 4.3 shows the mean μ, standard deviation σ and the maximum time of execution for the three functions on the LG Nexus 5 and the LG Nexus 5X. The API interception function using the Xposed framework averaged about $\mu = 0.2ms$ on both the phones, which is negligible from a usage perspective. The add audit event function of

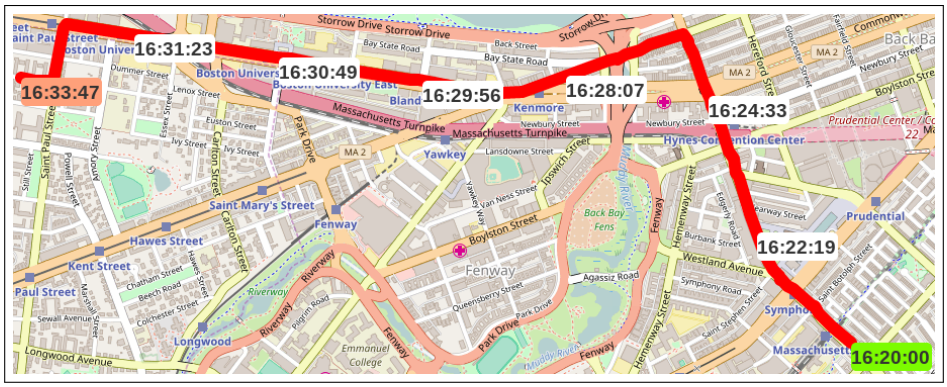
Table 4.3: Results of the Performance analysis of the MATRIX framework for 2 smartphones.

Phone	Service	Mean (μ)	Std (σ)	Max
LG Nexus 5	Xposed Framework	0.2 ms	0.3 ms	17.1 ms
	Add Audit Event	4.3 ms	3.8 ms	67.1 ms
	Update Location	11.1 ms	7.7 ms	87.6 ms
LG Nexus 5X	Xposed Framework	0.2 ms	0.15 ms	5.7 ms
	Add Audit Event	3.2 ms	1.6 ms	26.8 ms
	Update Location	5.7 ms	1.5 ms	16.0 ms

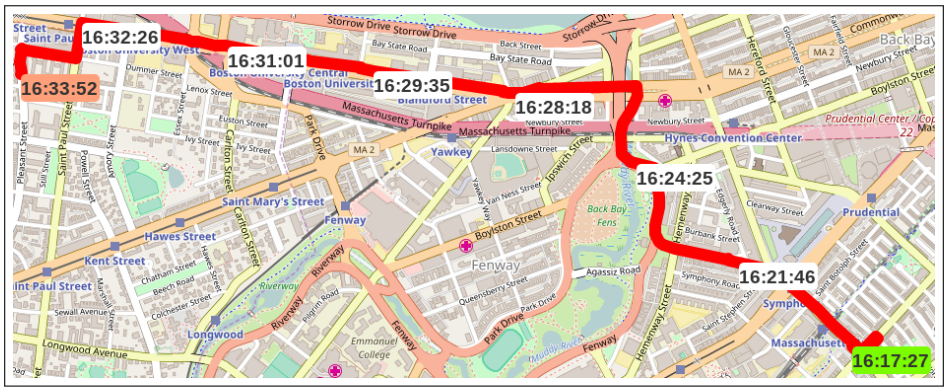
the PrivoScope service had a low μ for both the phones (4.3ms and 3.2ms, resp), and its performance was also acceptable. The location provider function of the Synthetic Location service had a relatively higher μ and σ for the Nexus 5 ($\mu = 11.1ms$, $\sigma = 7.7ms$). We believe this overhead is due to database lookups performed by the service to check the location preferences for the app. Overall, the entire system can run with an average overhead of 15.6ms on the Nexus 5 and 9.1ms on the Nexus 5X which should have a negligible impact on the user experience. The sum of worst case performances overhead at 171.8ms on the Nexus 5 should also not affect user experience since such overhead occurs rarely.

4.6.3 Detection of Synthetic Trajectories by Regular Users

To evaluate this metric, we conducted two separate user studies; one comprising of a group of 12 students from the university and the other comprising of 100 users from Amazon Mechanical Turk [120]. The survey asked the users to visually analyze 20 driving trajectories and label them as either 'Real' or 'Synthetic' based of their observations about the trajectory. Figure 4.11a shows one example of a real driving route and Figure 4.11b shows the corresponding generated synthetic route given to the users of the study. The green marker marks the start location, the white markers are 500m apart, and the red marker marks the stop location. These markers display the time the vehicle was at the given location. Note that the two routes follow a different path to the destination, however, they have similar travel times. This difference is because the Dijkstra's algorithm finds the fastest route between source and destination from the graph. We verified that



(a) Real Driving Route

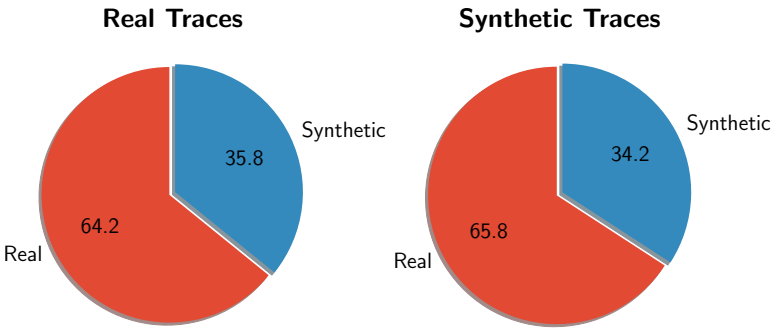


(b) Generated Synthetic Trajectory

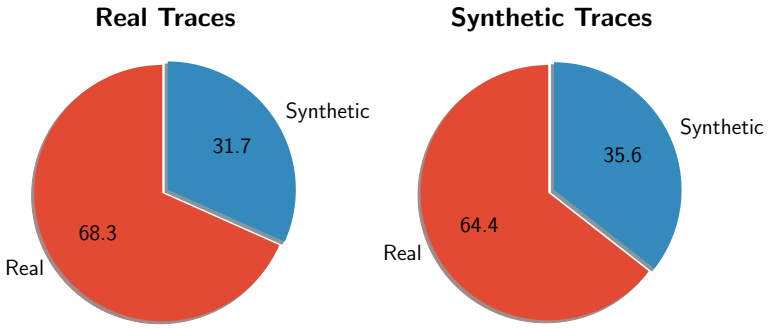
Figure 4.11: An example of the similarity between a real route and a generated synthetic route for the same start and end locations, and similar departure time.

the synthetic route is the same as recommended route by Google Maps for the endpoints.

The intuition behind two studies was to understand the results from two perspectives; one of users who know the area very well and other of users unaware of the area. The university area was chosen so that the students were aware of its traffic congestions. The trajectories were created as follows: First, we drove 10 unique routes close to the university area, each starting and ending at different locations and times of the day. Each route can be represented as $R = [n_1, \ldots, n_L]$, where n is a node, and L is the number of nodes in the route. Each node n_i is attributed with timing and geographic information $n_i = (t_i, \text{Loc}(n_i))$, where t_i is the timestamp, and $\text{Loc}(n_i)$ is the node's geographic coordinates. Then, we generated 10 synthetic routes similar to the 10 real routes using the



(a) Results from University Students User Study



(b) Results from Amazon Mechanical Turk User Study

Figure 4.12: Cumulative results of the user study for real driving and generated synthetic trajectories. The results show confusion among the users regarding validity of the trajectories in both the studies.

timestamp of the first node (i.e., t_1) and geographic coordinates of the end nodes (i.e., Loc(n_1) and Loc(n_L)) for each route R. The trajectories were shuffled so they appear in a random order. For the mechanical turk study, we also added three very noisy trajectories which looked obviously synthetic to find users who did not take the survey seriously.

Figures 4.12a and 4.12b show the cumulative results of the university study and the mechanical turk study. For the mechanical turk study, they show the results only for 54 users who correctly detected all the obviously noisy traces.

University Students Study: For the real trajectories, $\approx$ 64.2% of the trajectories were labeled as 'Real' and the rest were labeled as 'Synthetic'. For the synthetic trajectories,

Table 4.4: Cumulative results of the User Study on Amazon Mechanical Turk sorted by the number of noisy trajectories correctly labeled.

Noisy	Surveyors	Real Trajectories		Synthetic Trajectories	
		Real	Synthetic	Real	Synthetic
0	100	65.1%	34.9%	66.0%	34.0%
1	91	65.4%	34.6%	65.9%	34.1%
2	72	65.7%	34.3%	65.4%	34.6%
3	54	68.3%	31.7%	64.4%	35.6%

$\approx$ 65.8% of the trajectories were labeled as 'Real' and the rest were labeled as 'Synthetic'. Note that more users of this study confused the 'Synthetic' trajectories to be 'Real'.

Amazon Mechanical Turk Study: For the real trajectories, $\approx$ 68.3% of the trajectories were labeled as 'Real' and the rest were labeled as 'Synthetic'. For the synthetic trajectories, $\approx$ 64.4% of the trajectories were labeled as 'Real'. The above results are for the 54 users who detected all the obviously noisy trajectories. Table 4.4 shows the cumulative results of the mechanical turk study based on the number of noisy trajectories detected by the users. We can see that the results are not significantly different even for all 100 users, however, more users labeled the 'Synthetic' routes as 'Real'.

The results indicate that it was difficult for the users to differentiate synthetic and real driving trajectories. There was confusion in both groups regarding the validity of the trajectories. Evaluating individual routes, we saw that this confusion applied to each trajectory as none of them were labeled as 'Real' or 'Synthetic' unanimously by all users.

4.6.4 Detection of Synthetic Trajectories by Popular Apps

We evaluated this metric using 10 popular location based apps (listed in Table 4.5) from the Google Play Store. These apps rely heavily on location data to provide the expected functionality to their users. The test was performed by feeding these apps three different types of synthetic location data and monitoring their behavior. In test 1 **(Synthetic)**, the synthetic trajectories were generated using the techniques described in Section 4.5.3. In test 2 **(HS)**, the trajectories from test 1 were time compressed by a factor of 5 such that

Table 4.5: Results of the Synthetic Trajectories detection test on 10 popular Android apps that rely on location data.

App Name	Category	Rating	Synthetic	High Speed (HS)	HS + Teleport (HS + T)
Ingress	Adventure Game	4.3	✓	Detected	Detected
Pokémon Go	Adventure Game	4.1	✓	✓	✓
Geocaching	Health & Fitness	4.0	✓	✓	✓
Glympse	Social	4.5	✓	✓	✓
Family Locator	Lifestyle	4.4	✓	✓	✓
happn	Lifestyle	4.5	✓	✓	✓
Yelp	Travel & Local	4.3	✓	✓	✓
Foursquare	Food & Drink	4.1	✓	✓	✓
Waze	Maps & Navigation	4.6	✓	✓	Unstable
Google Maps	Travel & Local	4.3	✓	✓	Unstable

the user appeared to move 5 times faster (e.g., at $300km/h$ in a $60km/h$ speed zone). In test 3 **(HS + T)**, the trajectories from test 2 were perturbed by large noises ($\approx 1000m$) such that the user appeared to teleport to different locations very quickly. The expected results was that apps that detect fake location should be able to easily detect the **HS** and **HS + T** trajectories, but not the **Synthetic** trajectories.

Table 4.5 shows the results of the three tests for our test apps. None of the apps were able to detect synthetic locations in the **Synthetic** trajectories test. Even for **HS** and **HS + T** trajectories, with the exception of Ingress, none of the other apps detected the presence of high speed and noisy synthetic locations. Ingress did not ban us from playing the game, however, it denied points when it detected that the user was moving too fast or teleporting. Pokémon Go is also known to ban users, however, we did not get banned during our tests even after capturing many Pokémons using the noisy data. This is likely because the ban threshold is set to high to prevent users from going to a higher level by cheating. All the remaining apps kept performing their functions without detecting the presence of the synthetic data. Note that Waze and Google Maps navigation operated properly for **HS** but became unstable for **HS + T**, which was expected as they constantly updated the routing information based on the teleported locations.

Table 4.6: Results of the Machine Learning algorithms evaluation showing the 'Real' and 'Synthetic' prediction accuracy.

Algorithm	**Real Trajectories**		**Synthetic Trajectories**	
	Real	**Synthetic**	**Real**	**Synthetic**
Decision Trees	53%	47%	38%	62%
Random Forest	61%	39%	37%	63%
Nearest Neighbor	50%	50%	43%	57%
10 Nearest Neighbor	49%	51%	43%	57%
Naive Bayes	86%	14%	86%	14%
Neural Networks	95%	5%	96%	4%
SVM	5%	95%	3%	97%

These observations indicate that popular apps that rely on location data fail to check the validity of the received data. Some of these apps (Ingress, Pokémon Go, Foursquare and Google Maps) implement checks to detect whether the `MockLocationProvider` [121] is enabled on the device. The only app that checked location validity in our set was Ingress, and it was unable to detect any discrepancies in the synthetic trajectories generated by our system.

4.6.5 Detection of Synthetic Trajectories by Machine Learning Algorithms

We evaluated this metric using the 400 routes collected for analyzing user driving behavior (cf. Section 4.5.3). These set of routes were labeled as 'Real' classifier. For each real route, a corresponding synthetic route was generated using the real route's departure time, and start and end locations. These set of routes were labeled as 'Synthetic' classifier. The following 9 features were extracted from both set of routes for training the machine learning models: *max and min acceleration*, *mean and standard deviation of accelerations*, *mean and standard deviation of absolute accelerations*, *maximum speed*, *idle time* and *distance traveled*. The models were built and the predictions were averaged over 1000 iterations. In each iteration, 90% of the dataset from each set were randomly chosen for training data, and the remaining 10% from each set were test data.

Table 4.6 shows the list of algorithms that were tested, and their prediction accuracies

for the 'Real' and 'Synthetic' test trajectories. Note that in our context, the ideal results should be a 50-50 split, i.e., 50% of 'Real' routes are predicted as 'Synthetic' and 50% of 'Synthetic' routes are predicted as 'Real'. We can observe that most algorithms (except Decision Trees and Random Forest) have an average prediction accuracy close to 50%. Three of those algorithms (Naive Bayes, Neural Network and SVM) display results biased towards one of the two classifiers implying that the models had difficulty predicting the correct classifier and defaulted to one classifier. The Decision Trees and Random Forest models could detect $\approx$ 63% of the 'Synthetic' trajectories as synthetic. However, these numbers also do not signify large detection rate for our synthetic trajectories. We must note that this evaluation is preliminary as 400 routes do not suffice for these algorithms to build generalized models from training data, and the models may be subject to overfitting. We intend to extend our dataset in the future to incorporate more routes and run this evaluation again for more generalized models.

4.7 Conclusion

We presented the design and evaluation of MATRIX, a framework and system that addresses some current privacy protection weaknesses in Android, and provides users with a tool to analyze how apps access their private information as well as the capability to provide obfuscated / synthetic data to untrusted apps. Synthetic, yet realistic, mobility trajectories have the potential to reduce privacy leaks and enable the understanding of how users' location information is exploited by mobile apps. We demonstrated that MATRIX is portable to most Android devices globally, has low-overhead, is reliable, and generates privacy-preserving synthetic trajectories that are difficult to differentiate from real mobility trajectories by an adversary.

Chapter 5

Future Work

The two attacks described in this work demonstrate a worst-case attack performance, with no a priori information about the victim. We believe that this performance can be significantly improved if more information about the victim is collected over time. The MATRIX framework is a preliminary step in implementing an extensible mitigation framework open to the research community and users globally. In this section, we discuss some of the possible future extensions of this work.

Single-stroke Language-Agnostic Keylogging: This attack demonstrated the feasibility of inferring single keystrokes and does not use any lexical properties of languages. The focus was on inferring passwords, PINs and credit card numbers from sensitive apps in a single attempt. In reality, users may use these apps multiple times and enter their credentials each time to get access. The Gyroscope and Microphone recordings of the keystrokes from the multiple recordings can be averaged together to offset the noise and improve the attack performance. Moreover, multiple predictions of the same keystroke for a specific index of a password increases confidence in that keystroke and helps an adversary focus on the other indexes. We intend to study the impact due to collecting multiple samples from the victim on the attack performance in the future. The attack can also be extended to infer user typed sentences such as sensitive emails. This would require a priori knowledge about the language a victim uses for typing, however, this can be easily inferred from the geographic location of the users (e.g., from their IP address). This attack can be implemented by combining our keystroke inference meta-algorithm with Natural Language Processing (NLP) techniques. We also intend to study this extension of our attack in the future.

Inferring User Routes and Locations: This attack demonstrated the feasibility of inferring a victim's vehicular routes with no a priori information about them. Like the

previous attack, the focus was on inferring the route in a single attempt from a single recording of the sensors. We also assumed that all victim routes are equiprobable. In reality, the probability of certain routes occurring are much higher than the others. For example, a victim who works in a office will have a higher probability of driving from home to office, and office to home during the weekdays. Such travel history information can also be built up over time to improve the attack performance. One potential technique for inference in such cases could be to detect significant routes by clustering very similar routes together. The turns and curvatures of all routes in a cluster can be averaged together to obtain more accurate turn and curvature information. Another potential technique could be to update the search algorithm to maximize the likelihood of finding the starting vertex. We intend to study the above two techniques in the future.

The simulations for all cities in our set were performed using the same attack configuration of scoring weights and filtering thresholds. We intend to perform a rigorous analysis of these parameters to determine the ideal settings for different road networks. We believe this will significantly improve the attack performance for every city.

Extensions to the MATRIX Framework: The current MATRIX framework implementation focuses on location based attacks that exploit the location and sensor APIs on Android smartphones. Other attack vectors such as Wi-Fi and telephony APIs can also be exploited to track users. Some other attack vectors such as the Camera and Microphone APIs can also be exploited to snoop on user activities. We intend to extend our PrivoScope service to audit these APIs and provide a framework and system that gives users a bigger picture of how apps access their private data. For synthetic location generation, we focused mostly on vehicular routes in this work and implemented a simple linear model for simulating user walks. Some of our next steps will be to support realistic pedestrian mobility-patterns and incorporate other design mechanisms for generating mobility patterns (e.g., [122]). We will also be working on updating our current synthetic location generation technique such that it provides better resiliency against machine learning detection algorithms. We also intend to use this framework in the future to feed honey-synthetic data to apps and study how apps misuse users location information.

Bibliography

[1] FTC. Android flashlight app developer settles FTC charges it deceived consumers. https://www.ftc.gov/news-events/press-releases/2013/12/android-flashlight-app-developer-settles-ftc-charges-it-deceived, December 2013. Accessed: November, 2015.

[2] Kassem Fawaz and Kang G. Shin. Location privacy protection for smartphone users. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 239–250. ACM, 2014.

[3] Senate Judiciary Committee. S.2171 - Location Privacy Protection Act of 2014. https://www.congress.gov/bill/113th-congress/senate-bill/2171, 2014.

[4] OpenStreetMap. OpenStreetMap Project. https://www.openstreetmap.org/.

[5] Google Inc. Google Maps Directions API. https://developers.google.com/maps/documentation/directions/, 2017.

[6] Liang Cai and Hao Chen. Touchlogger: Inferring keystrokes on touch screen from smartphone motion. In *Proceedings of the 6th USENIX Conference on Hot Topics in Security*, HotSec'11.

[7] Zhi Xu, Kun Bai, and Sencun Zhu. Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12.

[8] Adam J. Aviv, Benjamin Sapp, Matt Blaze, and Jonathan M. Smith. Practicality of accelerometer side channels on smartphones. In *Proceedings of the Annual Computer Security Applications Conference*, ACSAC '12.

[9] Emmanuel Owusu, Jun Han, Sauvik Das, Adrian Perrig, and Joy Zhang. Acces-

sory: Password inference using accelerometers on smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, HotMobile '12.

[10] Emiliano Miluzzo, Alexander Varshavsky, Suhrid Balakrishnan, and Romit Roy Choudhury. Tapprints: Your finger taps have fingerprints. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12.

[11] Robert Templeman, Zahid Rahman, David Crandall, and Apu Kapadia. PlaceRaider: Virtual theft in physical spaces with smartphones. In *The 20th Annual Network and Distributed System Security Symposium*, NDSS '13.

[12] Anindya Maiti, Murtuza Jadliwala, Jibo He, and Igor Bilogrevic. (smart)watch your taps: Side-channel keystroke inference attacks using smartwatches. In *Proceedings of the 2015 ACM International Symposium on Wearable Computers*, ISWC '15. ACM, 2015.

[13] Philip Marquardt, Arunabh Verma, Henry Carter, and Patrick Traynor. (sp)iphone: Decoding vibrations from nearby keyboards using mobile phone accelerometers. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11.

[14] Thomas G. Dietterich. Ensemble methods in machine learning. In *Proceedings of the First International Workshop on Multiple Classifier Systems*, MCS '00.

[15] Euclid Analytics. Privacy statement. `http://euclidanalytics.com/privacy/statement/`. Accessed: May, 2015.

[16] Siraj Datoo. This recycling bin is following you. `http://qz.com/112873/this-recycling-bin-is-following-you/`, Quartz, August 2013. Accessed: May, 2015.

[17] Zachary M. Seward and Siraj Datoo. City of london halts recycling bins tracking phones of passers-by. `http://qz.com/114174/city-of-london-halts-recycling-bins-tracking-phones-of-passers-by/`, Quartz, August 2013. Accessed: May, 2015.

[18] Lee Hutchinson. iOS 8 to stymie trackers and marketers with mac address randomization. http://arstechnica.com/apple/2014/06/ios8-to-stymie-trackers-and-marketers-with-mac-address-randomization/, June 2014. Accessed: May, 2015.

[19] Yan Michalevsky, Aaron Schulman, Gunaa Arumugam Veerapandian, Dan Boneh, and Gabi Nakibly. Powerspy: Location tracking using mobile device power analysis. In *Proceedings of the 24th USENIX Conference on Security Symposium*, pages 785–800, Washington, D.C., August 2015. USENIX Association.

[20] Xiaoyong Zhou, Soteris Demetriou, Dongjing He, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, Carl A. Gunter, and Klara Nahrstedt. Identity, location, disease and more: inferring your secrets from android public resources. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, CCS '13.

[21] Jun Han, E. Owusu, L.T. Nguyen, A. Perrig, and J. Zhang. Accomplice: Location inference using accelerometers on smartphones. In *Communication Systems and Networks (COMSNETS), 2012 Fourth International Conference on*, pages 1–9, Jan 2012.

[22] Sarfraz Nawaz and Cecilia Mascolo. Mining users' significant driving routes with low-power sensors. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, SenSys '14, pages 236–250. ACM, 2014.

[23] Miguel E. Andrés, Nicolás E. Bordenabe, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. Geo-indistinguishability: Differential privacy for location-based systems. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, 2013.

[24] Nicolás E. Bordenabe, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. Optimal geo-indistinguishable mechanisms for location privacy. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, 2014.

[25] Reza Shokri, George Theodorakopoulos, Carmela Troncoso, Jean-Pierre Hubaux, and Jean-Yves Le Boudec. Protecting location privacy: Optimal strategy against

localization attacks. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012.

[26] Y. Wang, Dingbang Xu, Xiao He, Chao Zhang, Fan Li, and B. Xu. L2p2: Location-aware location privacy protection for location-based services. In *2012 Proceedings IEEE INFOCOM*, March 2012.

[27] C. A. Ardagna, M. Cremonini, S. De Capitani di Vimercati, and P. Samarati. An obfuscation-based approach for protecting location privacy. *IEEE Transactions on Dependable and Secure Computing*, Jan 2011.

[28] Baik Hoh and Marco Gruteser. Preserving privacy in gps traces via uncertainty-aware path cloaking. In *In Proceedings of ACM CCS 2007*, 2007.

[29] Ryo Kato, Mayu Iwata, Takahiro Hara, Akiyoshi Suzuki, Xing Xie, Yuki Arase, and Shojiro Nishio. A dummy-based anonymization method based on user trajectory with pauses. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '12, 2012.

[30] H. Kido, Y. Yanagisawa, and T. Satoh. An anonymous communication technique using dummies for location-based services. In *ICPS '05. Proceedings. International Conference on Pervasive Services, 2005.*, July 2005.

[31] Hua Lu, Christian S. Jensen, and Man Lung Yiu. Pad: Privacy-area aware, dummy-based location privacy in mobile services. In *Proceedings of the Seventh ACM International Workshop on Data Engineering for Wireless and Mobile Access*, MobiDE '08, 2008.

[32] Akiyoshi Suzuki, Mayu Iwata, Yuki Arase, Takahiro Hara, Xing Xie, and Shojiro Nishio. A user location anonymization method for location based services in a real environment. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '10, 2010.

[33] T. H. You, W. C.f Peng, and W. C. Lee. Protecting moving trajectories with dummies. In *2007 International Conference on Mobile Data Management*, May 2007.

[34] A. Pingley, N. Zhang, X. Fu, H. A. Choi, S. Subramaniam, and W. Zhao. Protection of query privacy for continuous location based services. In *2011 Proceedings IEEE INFOCOM*, April 2011.

[35] Baik Hoh, M. Gruteser, Hui Xiong, and A. Alrabady. Enhancing security and privacy in traffic-monitoring systems. *IEEE Pervasive Computing*, Oct 2006.

[36] R. Shokri, G. Theodorakopoulos, J. Y. Le Boudec, and J. P. Hubaux. Quantifying location privacy. In *2011 IEEE Symposium on Security and Privacy*, May 2011.

[37] Hui Zang and Jean Bolot. Anonymization of location data does not work: A large-scale measurement study. In *Proceedings of the 17th Annual International Conference on Mobile Computing and Networking*, MobiCom '11, 2011.

[38] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock android. In *24th USENIX Security Symposium (USENIX Security 15)*, Washington, D.C., 2015. USENIX Association.

[39] B. Krupp, N. Sridhar, and W. Zhao. Spe: Security and privacy enhancement framework for mobile devices. *IEEE Transactions on Dependable and Secure Computing*, 2015.

[40] 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*, HotMobile '11, New York, NY, USA, 2011. ACM.

[41] Igor Bilogrevic, Kvin Huguenin, Berker Agir, Murtuza Jadliwala, Maria Gazaki, and Jean-Pierre Hubaux. A machine-learning based approach to privacy-aware information-sharing in mobile social networks. *Pervasive and Mobile Computing*, 25, 2016.

[42] Yuvraj Agarwal and Malcolm Hall. Protectmyprivacy: Detecting and mitigating privacy leaks on ios devices using crowdsourcing. In *Proceeding of the 11th Annual*

International Conference on Mobile Systems, Applications, and Services, MobiSys '13, New York, NY, USA, 2013. ACM.

[43] Bin Liu, Mads Schaarup Andersen, Florian Schaub, Hazim Almuhimedi, Shikun (Aerin) Zhang, Norman Sadeh, Yuvraj Agarwal, and Alessandro Acquisti. Follow my recommendations: A personalized privacy assistant for mobile app permissions. In *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*, Denver, CO, 2016. USENIX Association.

[44] B. Deva, S. R. Garzon, and S. Schnemann. A context-sensitive privacy-aware framework for proactive location-based services. In *2015 9th International Conference on Next Generation Mobile Applications, Services and Technologies*, Sept 2015.

[45] Kassem Fawaz, Huan Feng, and Kang G. Shin. Anatomization and protection of mobile apps' location privacy threats. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, 2015.

[46] 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 realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, Berkeley, CA, USA, 2010. USENIX Association.

[47] Mingshen Sun, Tao Wei, and John C.S. Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, New York, NY, USA, 2016. ACM.

[48] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu. Effective real-time android application auditing. In *2015 IEEE Symposium on Security and Privacy*, May 2015.

[49] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. Taming information-stealing smartphone applications (on android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, TRUST'11, Berlin, Heidelberg, 2011. Springer-Verlag.

[50] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, New York, NY, USA, 2014. ACM.

[51] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, New York, NY, USA, 2014. ACM.

[52] Michael Backes, Sven Bugiel, Erik Derr, Sebastian Gerling, and Christian Hammer. R-droid: Leveraging android app analysis with static slice optimization. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, New York, NY, USA, 2016. ACM.

[53] Kangjie Lu, Zhichun Li, Vasileios P. Kemerlis, Zhenyu Wu, Long Lu, Cong Zheng, Zhiyun Qian, Wenke Lee, and Guofei Jiang. Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In *The Network and Distributed System Security Symposium*, NDSS '15, 2015.

[54] Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd Millstein. Dr. android and mr. hide: Fine-grained permissions in android applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, New York, NY, USA, 2012. ACM.

[55] Benjamin Davis and Hao Chen. Retroskeleton: Retrofitting android apps. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, New York, NY, USA, 2013. ACM.

[56] 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*, CCS '11, New York, NY, USA, 2011. ACM.

[57] Suwen Zhu, Long Lu, and Kapil Singh. Case: Comprehensive application security enforcement on cots mobile devices. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16. ACM, 2016.

[58] Yihang Song and Urs Hengartner. Privacyguard: A vpn-based platform to detect information leakage on android devices. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '15, 2015.

[59] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David R. Choffnes. Recon: Revealing and controlling privacy leaks in mobile network traffic. *CoRR*, abs/1507.00255, 2015.

[60] A. Pham, K. Huguenin, I. Bilogrevic, I. Dacosta, and J. P. Hubaux. Securerun: Cheat-proof and private summaries for location-based activities. *IEEE Transactions on Mobile Computing*, 15(8), Aug 2016.

[61] V. Bindschaedler and R. Shokri. Synthesizing plausible privacy-preserving location traces. In *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016.

[62] Ashwin Machanavajjhala, Daniel Kifer, John Abowd, Johannes Gehrke, and Lars Vilhuber. Privacy: Theory meets practice on the map. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE '08, 2008.

[63] John Krumm. Realistic driving trips for location privacy. In *International Conference on Pervasive Computing*. Springer, 2009.

[64] Richard Chow and Philippe Golle. Faking contextual data for fun, profit, and privacy. In *Proceedings of the 8th ACM Workshop on Privacy in the Electronic Society*, WPES '09, 2009.

[65] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12.

[66] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*.

[67] Google Inc. Android software development kit (sdk). `http://developer.android.com/sdk/index.html`, 2014. Last accessed 02/27/2014.

[68] Google Inc. Android native development kit (sdk). `https://developer.android.com/tools/sdk/ndk/index.html`, 2014. Last accessed 02/25/2014.

[69] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*.

[70] Greg Welch and Gary Bishop. An introduction to the kalman filter. Technical report, 1995.

[71] Sky Mckinley and Megan Levine. Cubic spline interpolation.

[72] Ludmila I. Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms*. John Wiley and Sons, Inc., 2004.

[73] Liang Cai and Hao Chen. On the practicality of motion based keystroke inference attack. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST'12.

[74] Roman Schlegel, Kehuan Zhang, Xiao Yong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *The 18th Annual Network and Distributed System Security Symposium*, NDSS '11.

[75] Shu Lin and Daniel J. Costello. *Error Control Coding*. 2 edition, 2004.

[76] Nokia. HERE Map. https://maps.here.com/.

[77] H. Park, D. Ahn, T. Park, and K. G. Shin. Automatic identification of driver's smartphone exploiting common vehicle-riding actions. *IEEE Transactions on Mobile Computing*, 2017.

[78] Zhiyun Qian, Zhaoguang Wang, Qiang Xu, Z. Morley Mao, Ming Zhang, and Yi-Min Wang. You can run, but you can't hide: Exposing network location for targeted DoS attacks in cellular networks. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, February 2012.

[79] Denis F. Kune, John Koelndorfer, Nicholas Hopper, and Yongdae Kim. Location leaks over the GSM air interface. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, February 2012.

[80] Laurent Bindschaedler, Murtuza Jadliwala, Igor Bilogrevic, Imad Aad, Philip Ginzboorg, Valtteri Niemi, and Jean-Pierre Hubaux. Track me if you can: On the effectiveness of context-based identifier changes in deployed mobile networks. In *NDSS*. The Internet Society, 2012.

[81] Natalia Marmasse and Chris Schmandt. A user-centered location model. *Personal and Ubiquitous Computing*, 6(5-6):318–321, 2002.

[82] DonaldJ. Patterson, Lin Liao, Dieter Fox, and Henry Kautz. Inferring high-level behavior from low-level sensors. In AnindK. Dey, Albrecht Schmidt, and JosephF. McCarthy, editors, *UbiComp 2003: Ubiquitous Computing*, volume 2864 of *Lecture Notes in Computer Science*, pages 73–89. Springer Berlin Heidelberg, 2003.

[83] Daniel Ashbrook and Thad Starner. Using GPS to learn significant locations and predict movement across multiple users. *Personal Ubiquitous Comput.*, October 2003.

[84] DonaldJ. Patterson, Lin Liao, Krzysztof Gajos, Michael Collier, Nik Livic, Katherine Olson, Shiaokai Wang, Dieter Fox, and Henry Kautz. Opportunity knocks: A system to provide cognitive assistance with transportation services. In Nigel Davies, ElizabethD. Mynatt, and Itiro Siio, editors, *UbiComp 2004: Ubiquitous Computing*, volume 3205 of *Lecture Notes in Computer Science*, pages 433–450. Springer Berlin Heidelberg, 2004.

[85] Jong Hee Kang, William Welbourne, Benjamin Stewart, and Gaetano Borriello. Extracting places from traces of locations. In *Proceedings of the 2nd ACM International*

Workshop on Wireless Mobile Applications and Services on WLAN Hotspots, WMASH '04, 2004.

[86] Kari Laasonen, Mika Raento, and Hannu Toivonen. Adaptive on-device location recognition. In Alois Ferscha and Friedemann Mattern, editors, *Pervasive Computing*, volume 3001 of *Lecture Notes in Computer Science*, pages 287–304. Springer Berlin Heidelberg, 2004.

[87] Lin Liao, Donald J. Patterson, Dieter Fox, and Henry Kautz. Learning and inferring transportation routines. *Artificial Intelligence*, 171(5-6):311–331, April 2007.

[88] Lei Zhang, Jiangchuan Liu, Hongbo Jiang, and Yong Guan. Senstrack: Energy-efficient location tracking with smartphone sensors. *Sensors Journal, IEEE*, 13(10):3775–3784, Oct 2013.

[89] Xiaoyong Zhou, Soteris Demetriou, Dongjing He, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, Carl A. Gunter, and Klara Nahrstedt. Identity, location, disease and more: Inferring your secrets from android public resources. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 1017–1028. ACM, 2013.

[90] Jennifer R. Kwapisz, Gary M. Weiss, and Samuel A. Moore. Activity recognition using cell phone accelerometers. *SIGKDD Explor. Newsl.*, 12(2):74–82, March 2011.

[91] Seon-Woo Lee and K. Mase. Activity and location recognition using wearable sensors. *Pervasive Computing, IEEE*, 1(3):24–32, July 2002.

[92] K.Y. Kupeev and H.J. Wolfson. On shape similarity. In *Pattern Recognition, 1994. Vol. 1 - Conference A: Computer Vision amp; Image Processing., Proceedings of the 12th IAPR International Conference on*, volume 1, pages 227–231 vol.1, Oct 1994.

[93] R. Shokri, G. Theodorakopoulos, J.-Y. Le Boudec, and J.-P. Hubaux. Quantifying location privacy. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 247–262, May 2011.

[94] Android. The Android Source Code. `https://source.android.com/source/`, 2017.

[95] Xposed Framework. The Xposed Framework Source Code. `https://github.com/rovo89/XposedInstaller`, 2017.

[96] L. Zhang, Z. Cai, and X. Wang. Fakemask: A novel privacy preserving approach for smartphones. *IEEE Transactions on Network and Service Management*, June 2016.

[97] J. Ghosh, S. J. Philip, and C. Qiao. Sociological orbit aware location approximation and routing in manet. In *2nd International Conference on Broadband Networks, 2005.*, Oct 2005.

[98] K. Lee, S. Hong, S. J. Kim, I. Rhee, and S. Chong. Slaw: A new mobility model for human walks. In *IEEE INFOCOM 2009*, April 2009.

[99] Frans Ekman, Ari Keränen, Jouni Karvo, and Jörg Ott. Working day movement model. In *Proceedings of the 1st ACM SIGMOBILE Workshop on Mobility Models*, MobilityModels '08, 2008.

[100] D. Karamshuk, C. Boldrini, M. Conti, and A. Passarella. Human mobility models for opportunistic networks. *IEEE Communications Magazine*, December 2011.

[101] C. Tuduce and T. Gross. A mobility model based on wlan traces and its validation. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, March 2005.

[102] David R. Choffnes and Fabián E. Bustamante. An integrated mobility and traffic model for vehicular wireless networks. In *Proceedings of the 2Nd ACM International Workshop on Vehicular Ad Hoc Networks*, VANET '05, 2005.

[103] M. Kim, D. Kotz, and S. Kim. Extracting a mobility model from real user traces. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, April 2006.

[104] Qunwei Zheng, Xiaoyan Hong, Jun Liu, David Cordes, and Wan Huang. Agenda driven mobility modelling. *Int. J. Ad Hoc Ubiquitous Comput.*, December 2010.

[105] Xiaoyan Hong, Mario Gerla, Guangyu Pei, and Ching-Chuan Chiang. A group mobility model for ad hoc wireless networks. In *Proceedings of the 2Nd ACM Inter-*

national Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems, MSWiM '99, 1999.

[106] Baochun Li. On increasing service accessibility and efficiency in wireless ad-hoc networks with group mobility. *Wirel. Pers. Commun.*, April 2002.

[107] Klaus Herrmann. Modeling the sociological aspects of mobility in ad hoc networks. In *Proceedings of the 6th ACM International Workshop on Modeling Analysis and Simulation of Wireless and Mobile Systems*, MSWIM '03, 2003.

[108] Mirco Musolesi and Cecilia Mascolo. A community based mobility model for ad hoc network research. In *Proceedings of the 2Nd International Workshop on Multi-hop Ad Hoc Networks: From Theory to Reality*, REALMAN '06, 2006.

[109] Yufei Tao, Christos Faloutsos, Dimitris Papadias, and Bin Liu. Prediction and indexing of moving objects with unknown motion patterns. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, 2004.

[110] J. Zhou, H. V. Leong, Q. Lu, and K. C. K. Lee. Optimizing update threshold for distance-based location tracking strategies in moving object environments. In *2007 IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, June 2007.

[111] Ouri Wolfson and Huabei Yin. *Accuracy and Resource Consumption in Tracking and Location Prediction*. 2003.

[112] Z. Ding, L. Guo, and X. Meng. Adaptive location update mechanism for network-constrained moving objects in changeful traffic conditions. In *2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware*, May 2009.

[113] Y. K. Huang, I. F. Su, L. F. Lin, and Y. C. Chung. Efficient processing of updates for moving objects with varying speed and direction. In *2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*, March 2013.

[114] Yuan-Ko Huang. Indexing and querying moving objects with uncertain speed and direction in spatiotemporal databases. *Journal of Geographical Systems*, Apr 2014.

[115] TeamWin. TeamWin - TWRP. https://twrp.me/about/, 2017.

[116] OpenStreetMap. OpenStreetMap Building Key. http://wiki.openstreetmap.org/wiki/Key:building, 2017.

[117] OpenStreetMap. OpenStreetMap Amenity Key. http://wiki.openstreetmap.org/wiki/Key:amenity, 2017.

[118] Android. Android Dashboards. https://developer.android.com/about/dashboards/index.html, 2017.

[119] Android. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html, 2017.

[120] Amazon. Amazon Mechanical Turk. https://www.mturk.com/mturk/welcome, 2017.

[121] Android. Android Mock Location Provider. https://developer.android.com/guide/topics/location/strategies.html\#MockData, 2017.

[122] Simon Oya, Carmela Troncoso, and Fernando Pérez-González. Back to the drawing board: Revisiting the design of optimal location privacy-preserving mechanisms. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, 2017.