Acoustic Sensing: Mobile Applications and Frameworks

by

Yu-Chih Tung

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2018

Doctoral Committee:

Professor Kang G. Shin, Chair
Professor Jason Flinn
Professor Peter Honeyman
Associate Professor Mark Newman

Yu-Chih Tung

yctung@umich.edu

ORCID iD: 0000-0003-4217-6752

ACKNOWLEDGEMENTS

Completing the Ph.D. degree is a long long journey. I have enjoyed my 5 years at the University of Michigan, but this thesis would not have been completed without the help and support of my advisor, family, colleagues, and friends. I would like to thank all of them.

First, I want to thank my advisor, Kang G. Shin, the most professional person that I have ever worked with. Thank you for working with me and giving me the freedom to explore whatever research topic I liked. I will always remember what you told me on my first day that you would not push me to pursue the degree, but instead, hold my hand and lead me to gain the necessary experience to become a Ph.D.

Second, I would like to express my gratitude to my family, especially my wife, Yu-Ting. Being a husband who knows nothing about cooking, I would not have survived even a single day without your support. You have always been with me throughout the 5 years, including my worst and best times. You also brought me the biggest gift during my study, our lovely daughter, Alison.

Last, I am highly indebted to all the members of the Real-Time Computing Laboratory and all of my friends at the University of Michigan. I not only appreciate every discussion that we had but also every moment that we hung out together. I would especially like to thank Kassem and Arun. You spent an enormous amount of time with me to shape my ideas, clear my mind, and coach my English. I would also like to thank Dongyao for being my best friend at Michigan. I will always remember our Saturday basketball game.

My thanks and appreciations also go to all the people who have willingly helped me out over the last 5 years. I would also like to acknowledge the support of the National Science Foundation under Grant No. CNS-1646130.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

Acoustic sensing has attracted significant attention from both academia and industry due to its ubiquity. Since smartphones and many IoT devices are already equipped with microphones and speakers, it requires nearly zero additional deployment cost. Acoustic sensing is also versatile. For example, it can detect obstacles for distracted pedestrians (`BumpAlert`), remember indoor locations through recorded echoes (`EchoTag`), and also understand the touch force applied to mobile devices (`ForcePhone`).

In this dissertation, we first propose three acoustic sensing applications, `BumpAlert`, `EchoTag`, and `ForcePhone`, and then introduce a cross-platform sensing framework called `LibAS`. `LibAS` is designed to facilitate the development of acoustic sensing applications. For example, `LibAS` can let developers prototype and validate their sensing ideas and apps on commercial devices without the detailed knowledge of platform-dependent programming. `LibAS` is shown to require less than 30 lines of code in Matlab to implement the prototype of ForcePhone on Android/iOS/Tizen/Linux devices.

CHAPTER I

Introduction

Sensing with inaudible sounds, also known as acoustic sensing, has drawn considerable attention from both academia and industry. Research has shown that it can identify human movements [58, 62, 63, 79, 93, 124], augment device interactions [56, 81, 92, 100, 132], or provide context-aware computation [88, 109, 116, 116]. With the support of acoustic sensing, useful functions can be implemented to enhance the capability of users to interact with their devices. Since most mobile devices already have microphones and speakers installed, acoustic sensing applications are backward compatible and incur only minimal cost to deploy. We first introduce three practical acoustic sensing applications that have been implemented over different platforms. We then extract the core components from the experience of building these three applications to design a general framework for the development of future acoustic sensing applications.

1.1 Acoustic Sensing

While "acoustic sensing" often refers to a general concept of applications based on sounds, in this dissertation, it refers to *active* acoustic sensing. As shown in Fig. 1.1, *active* acoustic sensing utilizes the sound generated by the sensing device and then builds applications by processing the reception of this generated sound. In contrast, *passive* acoustic sensing only utilizes background/environment sounds. Sometimes, a mixture of active and

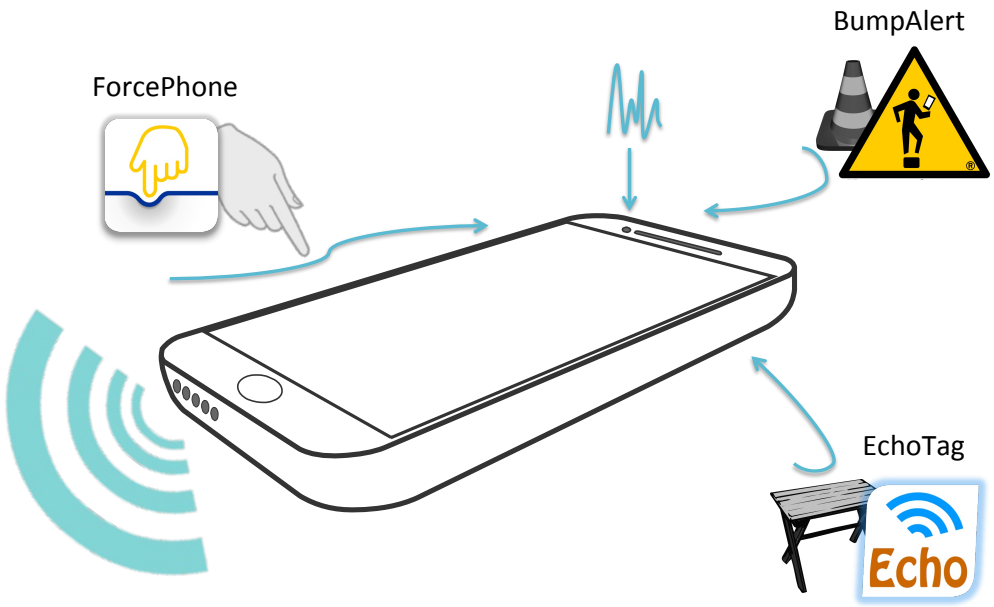


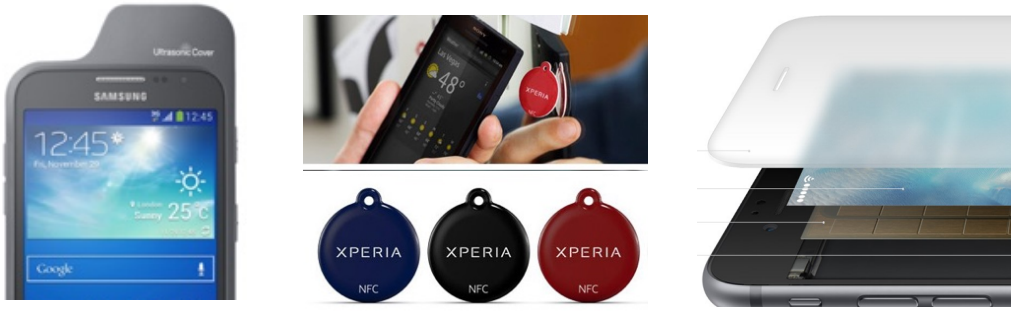
Figure 1.1: Active acoustic sensing. Sounds sent from the device speakers are picked up by the device microphones and analyzed to provide various functions.

passive acoustic sensing might be used. For example, we also utilize the background sounds as a reference to remove unwanted interference in our active acoustic sensing projects. In this thesis, we focus on active acoustic sensing because its capability to *actively* control different sensing signals can provide a wider spectrum to support various applications. For example, the *passively* recorded background noise (e.g., noise caused by heating or air conditioning systems) might always be identical in the same room, but the *actively* played sound and its echo in the same room will vary with location.

1.2 Thesis Statement

> Acoustic sensing enables backward-compatible and cross-platform applications on off-the-shelf hardware, thus introducing new sensing technology to users with minimal deployment cost.

We have implemented three such acoustic sensing applications. As shown in Fig. 1.1, `BumpAlert` [9, 120] identifies obstacles in front of distracted pedestrians through acoustic



(a) Obstacle detection (b) Location sensing (c) Force sensing

Figure 1.2: Dedicated sensor solutions. Adding dedicated sensors can easily realize novel sensing applications, but also incurs several deployment issues.

signals reflected by nearby objects. `EchoTag` [118] remembers a device's locations by listening to and analyzing environmental echoes. A location-based service can be triggered automatically when the phone is placed back at the pre-defined *echo* tag. `ForcePhone` [14, 119] creates a touch-force sensing interface based only on structure-borne propagated sounds, thus enriching human-computer interactions with minimal deployment cost.

1.3 Why Not Dedicated Sensors?

Adding dedicated sensors is the most straightforward way to build new sensing applications. For example, as shown in Fig. 1.2, an augmented sonar case can easily detect nearby obstacles, an NFC/RFID tag can remember tagged locations, and a proprietary Apple 3D Touch sensor can accurately estimate touch force. In general, these specially-designed sensors can achieve better system performance, like higher detection accuracy and fewer estimation errors. However, dedicated sensors also raise several practical issues in designing/deploying novel sensing applications:

- **Hardware cost.** Innovative new sensors are usually considered expensive. For example, Apple's 3D Touch Sensors (a thin layer connected by numerous force sensors underneath the screen) is estimated to cost around $7 to $9 in the iPhone7, which represents an 18% additional display cost [6]. This cost is estimated to be doubled in the new iPhone 8 due to design conflicts with the new OLED display [7].

- **Backward-compatibility.** Since most mobile devices are unable to add new hardware after they are manufactured, new dedicated sensors are usually only available in the latest flagship devices. Thus, it requires a relatively long time before users experience the novel sensing functions, which might reduce the chance to penetrate/-dominate the market.
- **Fragmented developing ecosystems.** The lack of new dedicated sensors on the devices that people already does not only delay the deployment of new sensing functions, but also fragments the market and incurs additional maintenance costs. For example, consider the fact that only some iPhones (and nearly 0% Android devices) have the force-sensing capability. How should app developers design a unified user interface by integrating the touch force?

Thus, this dissertation aims to answer the following three key questions:

1. Is it possible to build functionally equivalent novel sensing applications without adding new hardware?
2. How reliable would applications based on only acoustic signals and built-in sensors be, and what are their limits?
3. Could we have a framework to facilitate the development of such acoustic sensing applications?

1.4 Why Acoustic?

Other alternative signals could also be used to realize similar sensing functions. For example, WiFi (2.4GHz) signals have been widely studied to support various gesture detection or environment sensing systems [33,34,104,128,130]. Radio signals, like WiFi, are also more reliable and have less interference than acoustic signals due to their wider bandwidth and higher carrier frequency. Even though the WiFi chips have already been installed on many devices, these novel WiFi sensing systems are currently available only on special platforms, such as USRP [12], WARP [73], or even customized antenna systems [34]. Con-

Figure 1.3: Potentially supported devices. Many devices already have microphones/speakers installed, which might support acoustic sensing applications with minimal deployment cost.

sumer devices usually cannot support such WiFi sensing systems due to the lack of API to access/manipulate low-level signals from WiFi chips and also due to the lack of necessary computation resources to process wide-band WiFi signals in real time. To the best of our knowledge, only a few systems utilizing high-level information, such as received signal strength indication (RSSI) or channel state information (CSI), can be ported in commercial devices with a modified firmware [60]. Some sensing systems might not be integrated into handheld or IoT devices even in the near future due to the required displacement between sensing antennas [34].

On the other hand, sound is a natural interface for human conversations. Thus, devices like laptops, smartphones, watches, cams, and many IoT devices already have microphones and speakers. Moreover, real-time and low-level audio APIs are available for most operating systems, including Linux, Android, iOS, and Tizen, in order to support existing applications like VoIP, music streaming, or voice recognition. Fig. 1.3 shows the potential devices/platforms to which we might apply acoustic sensing applications easily. This

property enables the augmentation of new sensing technology by *repurposing* the built-in microphones and speakers, thus significantly reducing the deployment cost.

1.5 Challenges

Building sensing applications based on acoustic signals is challenging because built-in microphones/speakers are not designed for these purposes. Most microphones and speakers are usually optimized to send/receive high fidelity sounds only in the human-speakable/hearable range. For example, the frequency responses of microphones are not flat over all frequencies and the received signal strength at 20kHz can be even 30dB weaker than the sounds of 500Hz [81]. Imperfect hardware design also causes frequency leakage, thus making a noticeable noise at the start and end of supposedly "inaudible" signals. Many of these issues in our acoustic sensing applications are mitigated by utilizing signal processing techniques, integrating other built-in sensors, or tuning the device setting. Note that the solutions to these issues are usually platform-dependent, so realizing "ubiquitous" acoustic sensing applications on different platforms is even more challenging.

1.6 Contributions

In this dissertation, we propose the following three acoustic sensing applications which not only solve users' unmet needs but also help us understand the practicality of building novel applications based on sounds. Based on our experience of building these three applications, we propose a novel framework called `LibAS` which facilitates the prototyping and deploying of acoustic sensing designs on different platforms.

1.6.1 BumpAlert: Improving the safety of pedestrians only by use of smartphones

With the increasing emergency room visits of distracted pedestrians, `BumpAlert` is designed to reduce the risk of collision with dangerous objects when users walk around while

using smartphones. Specifically, BumpAlert utilizes the phone's built-in microphones/speakers to build a sonar-like system for detecting nearby obstacles. Since the built-in microphones/speakers are omni-directional, BumpAlert integrates the inertial sensors and also the camera, if necessary, to further improve the detection accuracy. In our measurements, BumpAlert enables smartphones to identify various objects 1–4m in front of users with higher than 90% accuracy and minimal false positives.

1.6.2 EchoTag: Remembering the tagged location by using environmental echoes

Unlike BumpAlert which aims to identify obstacles several meters away from the users, EchoTag listens to the echoes from the contacted surface and close-by objects to remember the phone's locations without any pre-installed infrastructure. Based on the profile of echoes reflected by cups, monitors, or tables, EchoTag can remember the tagged location accurately over time and then turn on specific functions/applications based on the tagged locations. For example, phone calls can be automatically turned off when the phone is placed at a tag near the bed or the user's behavior of taking medicine can be monitored by setting a tag near the medicine cabinet.

1.6.3 ForcePhone: Estimating touch force by structure-borne sounds

Unlike sensing through the reflected sounds, ForcePhone provides a force-sensing interface by listening to the sounds sent through the device body. We have utilized ForcePhone to build several high-level user interfaces, such as a hard-pressed option menu or a squeezable go-back navigation, which was previously built by adding proprietary sensors like Apple 3D Touch [5] or HTC squeezable frame [15]. We have shown that participants in our user study can easily accomplish simple tasks such as triggering the hidden option menu by applying touch force with ForcePhone.

1.7 A Cross-Platform Sensing Framework for Acoustic Sensing

During our development of acoustic sensing projects, we found many platform-dependent issues that are not directly related to the sensing algorithms but play a critical role in their deployment in real devices. For example, in Android, Automatic Gain Control (AGC) often prevents microphone from receiving stable acoustic responses and making the signal easily saturated. We could not turn off this AGC directly by using the default AGC API [1] in most devices (e.g., "false" is returned, indicating which is not implemented by the phone manufacturers). According to our experimental results, AGC should be disabled by tuning an argument named `AudioSource` to the recording API. iOS also has the similar hidden but crucial audio development settings. For instance, we need to change specific AudioSession settings to allow iOS devices to record and play audio at the same time.

Besides the aforementioned tweaks, different platforms also need different programming languages (e.g., Obj-C, Java, C) and programming paradigms (e.g., dedicated threads or callbacks for handling low-level audio ring buffers) to record audio in real time. These platform-dependent implementation details can be a roadblock for most researchers/engineers in developing acoustic sensing applications. In particular, this might happen to someone who is familiar with designing new computer-human interfaces or sensing functions but does not have experience in handling real-time acoustic sensing signals in commercial devices. For example, questions we often received about acoustic sensing include *"why is the audio not recorded in real time?"*, *"why is my 22kHz sensing sound audible?"*, or *"how can I debug my sensing algorithm on Android?"*. Thus, we extract/assemble the essential components and tools used in our implemented projects to build `LibAS` [17], which is an open-source framework to facilitate acoustic sensing developments.

`LibAS` has two major key features. First, it is cross-platform, i.e., Linux, iOS, Android, and Tizen. So, it can easily prototype and deploy the acoustic sensing applications on off-the-shelf smartphones, wearables, or IoT devices. Second, it has a remote testing mode that allows developers to focus on the design of key components like the sensing signals

and the processing of each received signal in Matlab. Specifically, developers can install our pre-built developing apps (called LibAS DevApp [20, 21]), which will receive sensing configurations and stream the sensed signals to a remote Matlab server in real time. In this "remote" mode, developers can quickly prototype their applications on real devices with zero knowledge of platform-dependent programming details.

Once developers are satisfied with the outcome/performance of their applications in remote mode, they can move the core processing logic (currently being handled in a Matlab callback function) to C/C++ for building standalone applications. Since `LibAS` provides a unified wrapper to connect this C/C++ callback function over platforms, developers can easily swap their sensing apps from remote mode to standalone mode. It is also possible to validate the C/C++ callback design in the Matlab remote mode by connecting our Matlab MEX wrapper [8]. Anytime developers find a bug or decide to add a new sensing function, they can easily swap back to the remote mode and enjoy the development with Matlab's strong analysis/visualization support. We choose this design pattern because we find it extremely difficult to manipulate and test acoustic signals directly on handheld devices, especially with system programming languages, like Java or Obj-C. This design choice is akin to the well-known WARP wireless testbeds [73], where the signal processing algorithm can be validated in a remote Matlab mode before it is integrated into the final FPGA implementation.

CHAPTER II

BumpAlert

2.1 Introduction

The risk of injury is reported to increase significantly when pedestrians are distracted by their use of smartphones while walking, i.e., *distracted walking*. Pedestrians are reported to notice 50% less environmental changes when they text on their phone while walking [85]. According to the number of emergency room visits reported in the United States in 2010, the rate of accidents due to pedestrians' smartphone uses has grown 10x in 5 years [95]. This accident rate is likely to be increasing rapidly with the increase of distracted smartphone users. Such accidents can also be severe; for example, people may walk distracted into the middle of the road and get knocked down by an oncoming car, or may bump into electric poles/trees causing head injuries. Recognizing this growing risk of cellphone users, in Chongqing, a sprawling city in central China, authorities have even set up a "cellphone lane" where people focusing on their phones can stroll without running into anyone or object not holding/using a phone [10]. Also, Taiwan government is about to establish a law to fine distracted pedestrians $10 to reduce the accident rate [13].

Reducing this risk by using the phone itself without requiring any additional sensors or infrastructural support has been drawing significant attention from both research and industry communities, but has not yet produced a complete solution. Some systems can identify cars by building an image classifier with the images of frontal cars, but cannot detect any

object beyond the cars [126]. Some others focus on preventing people from losing steps when they walk through the transitions between pathway and road [70]. While existing approaches address various specific aspects, their reliance on strong assumptions, like the shape or color of objects, prevent them from detecting general obstacles in the user's path. To fill this gap, we propose `BumpAlert`, which addresses an important but unexplored problem: "can commodity phones determine if the user is walking toward (dangerous) obstacles without assuming any prior knowledge of the objects?" Guaranteeing the elimination of all dangerous incidents is the ultimate goal of all safety systems but very hard, if not impossible, to achieve. Like most existing approaches, `BumpAlert` is an add-on phone function to enhance the safety of distracted pedestrians that aims to reduce the accident rate as much as possible at reasonable cost/overhead.

It is challenging to detect obstacles by utilizing only the built-in sensors in commodity phones. To achieve high detection accuracy at low computation and energy costs, we exploit several phone sensors. `BumpAlert` uses the phone's speakers and microphones to estimate the distance between the user and nearby objects, and also uses the phone's rear camera to validate the detected objects, only when necessary. Several novel algorithms are developed and implemented by exploiting these sensor inputs. For example, the false detections caused by omnidirectional phone speakers/microphones are removed by a novel *motion filter* that tracks the user's trajectory using inertial sensors. Also, the distances to obstacles can be estimated by a single camera without depth perception since the phone's height has already been determined by the `BumpAlert`'s acoustic detector. This paper makes several contributions in that `BumpAlert`

- is the first phone-based app to "actively" monitor the environment and alert distracted walkers in real time;
- relies only on sensors available in commodity smartphones, without requiring any specialized sensors;
- doesn't rely on any *a priori* knowledge of obstacles, thus offering a generic solution

applicable to a broad range of situations/environments; and

- consumes only a small fraction of resources, thus unaffecting users' experience in using their phones.

`BumpAlert` is implemented on the Android platform as an app using the OpenCV library and the Java Native Interface. Our evaluation results demonstrate its capability to detect objects with higher than 95% accuracy in typical outdoor/indoor environments and consume around 8% of battery power per hour while running as a mobile app.

We have evaluated `BumpAlert` with participants in a controlled environment. Although `BumpAlert` does not guarantee safety for all possible dangerous scenarios that distracted walkers might encounter, our user study shows that 71% of the participants agree that `BumpAlert`'s detection accuracy is useful and 86% of them are willing to accept `BumpAlert`'s energy cost for detecting dangerous obstacles with a high probability. A user-interface study based on Microsoft Kinect [65] also corroborates that a system displaying frontal obstacles can make distracted walkers feel safer and more confident. Moreover, 43% of the participants in this study have experienced bumping into objects during distracted walking, and 86% of them have heard others collided with obstacles. These results are consistent with studies done by others indicating the real danger of distracted walking. A demo video of `BumpAlert` can be found from [9].

The remainder of this paper is organized as follows. Section 2.2 summarizes the related work in accident prevention systems. Section 2.3 gives an overview of `BumpAlert` and Section 2.4 describes the implementation details. Sections 2.5 and 2.6 provide our experimental evaluation and user study, respectively. The paper concludes with Section 2.9.

2.2 Related Work

Obstacle detection and avoidance have been an active area of research [40,91,101,114] in the field of intelligent vehicles and robotics. Of particular interest is the active safety

systems deployed in automobiles to protect pedestrians. However, most of these systems require expensive devices such as RADAR, LIDAR, SONAR, and multiple cameras for detection of pedestrians and prediction of their movement. These solutions are not easily wearable by people as they are usually heavy or require advanced sensors, but they can be used as a basis for signal processing, especially for camera imaging and SONAR processing. Note some robots might use cheap sensors to detect obstacles, but these sensors are still specially designed for this purpose. For example, sonar sensors in robots are directional while phone speaker/microphone are not. Another direction of study focuses on detection of pedestrian(s) with the help of infrastructure, such as pre-deployed cameras at intersections [123]. However, the same cannot be assumed in mobile phone environments.

Instead of using advanced/expensive sensors, one can find and exploit various built-in sensors of smartphones. These include accelerometers which sense the phone's movement, gyroscopes which detect the phone's orientation, cameras and microphones which capture images and record sound in the surrounding environment. These sensors have prompted the development of various apps, such as indoor phone localization [86, 116], context-aware computing [96, 118], and human–computer interfaces [119, 124].

Although there exist a myriad of apps that exploit sensors to perform various functions on the phone, little has been done on distracted walkers' safety, despite its rapidly growing importance. A passive approach using the phone's rear camera was proposed in [2, 29] to take and display the frontal image as the background of apps. Since it is a passive solution, the user still has to be responsible for identifying and avoiding the obstacles shown on the screen of his phone. However, users usually focus on the task (e.g., playing a game) at hand and may not pay attention to the changes in the background of the app they are running. Moreover, there are also apps, such as games, that do not allow the change of background.

There are also other mobile apps that sense environments and provide active feedbacks. WalkSafe [126] is able to identify the frontal view of an (approaching) vehicle by using the phone's rear camera when pedestrians are making telephone calls while crossing the

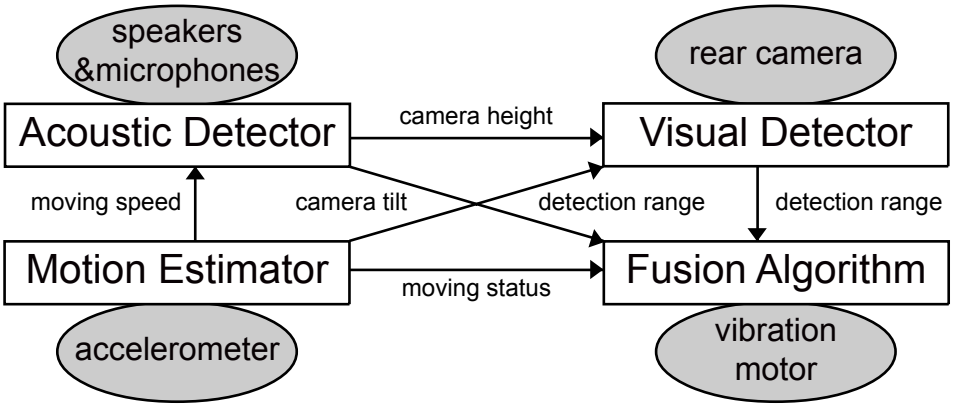


Figure 2.1: System blocks of BumpAlert. Multiple sensing components are utilized to optimize the detection performance.

road/street. LookUp [70] monitors the road transitions, such as the height change from a sidewalk to a street, by connecting inertial sensors mounted in shoes. Both apps target the scenarios parallel to `BumpAlert`, and it is possible to integrate `BumpAlert` with them to enhance pedestrians' safety. CrashAlert [65] targets the same scenario as ours, detecting obstacles when users are distracted walking. However, it mainly focuses on the design of walking user interface (WUI). The functionality of obstacle detection in CrashAlert is delegated to Microsoft Kinect, which is not available in commodity phones. In this paper, we explore how to detect and avoid objects in front of a distracted walker by using only the phone's built-in sensors and building and evaluating a mobile application called `BumpAlert`. Even though `BumpAlert` is unable to detect all dangerous situations (see Section 2.7), it has been shown to be able to detect most dangerous objects for distracted pedestrians, ranging from glass doors, sign boards, to a small parapet wall.

2.3 BumpAlert

As shown in Fig. 2.1, `BumpAlert` consists of four main components that interact with each other: (1) *acoustic detector* that uses sound to estimate the distances between the user and nearby objects; (2) *visual detector* that determines the presence of dangerous objects using the rear camera; (3) *motion estimator* that determines the user's walking

Algorithm 1 Acoustic Detection

Input: acoustic signal array at the n-th detection: S_n,
peak window: win_{peak}, walking speed: w, threshold coefficient: α
Output: detection result: D_{succ}

$S \leftarrow$ matched_filter(bandpass_filter(S_n))
$noise \leftarrow$ estimate_noise(S) & $thr \leftarrow \alpha(noise_{mean}+noise_{std})$
$P_n, D_n \leftarrow \phi$ & $peakMax, peakOffset \leftarrow 0$
for i from $win_{peak}/2$ to $len(S) - win_{peak}/2 - 1$ **do**
    $isPeak \leftarrow True$
    for j from $i - win_{peak}/2$ to $i + win_{peak}/2$ **do**
        if $S[j] > S[i]$ **then**
            $isPeak \leftarrow False$
            break
    if $isPeak$ and $S[i] > thr$ **then**
        $P_n \leftarrow P_n \cup i$
        if $S[i] > peakMax$ **then**
            $peakOffset \leftarrow i$
            $peakMax \leftarrow S[i]$
for $p \in P_n$ **do**
    $d = speed_{sound}(p - peakOffset)/(2rate_{sample})$
    $D_n \leftarrow D_n \cup d$
$D_{succ} \leftarrow$ motion_filter($D_n, D_{n-1}, \ldots, D_{n-\delta}, \delta, w$)
return D_{succ}

speed; and (4) *fusion algorithm* that combines information from all the other components and generates an alert for the user when a dangerous object is detected nearby.

2.3.1 Acoustic Detector

The acoustic detector borrows ideas from sonar sensors for object detection. The speaker sends 10 periods of a sine wave at a frequency of 11,025Hz and picks up its reflections through the phone's microphones. In order to make `BumpAlert` compatible with most commodity smartphones, the signal sent is sampled at 44.1kHz and two consecutive signals are transmitted with a 100ms separation to differentiate their reflections at the microphones/receivers. Note that this setting is designed to be widely supported by commodity phones and can be adopted widely as phone hardware improves. For example, Section 2.8 describes the extended setting designed for Galaxy Note4, which can provide reasonably good detection accuracy with *inaudible* sound.

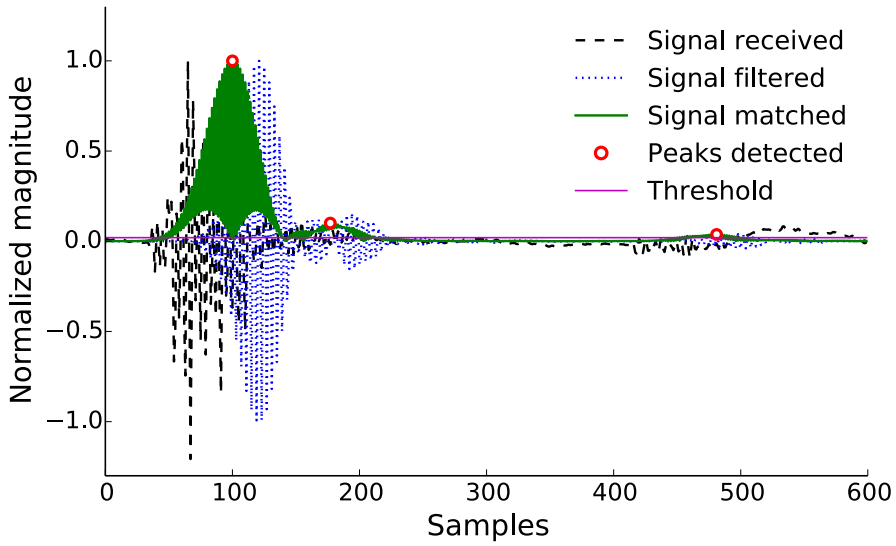
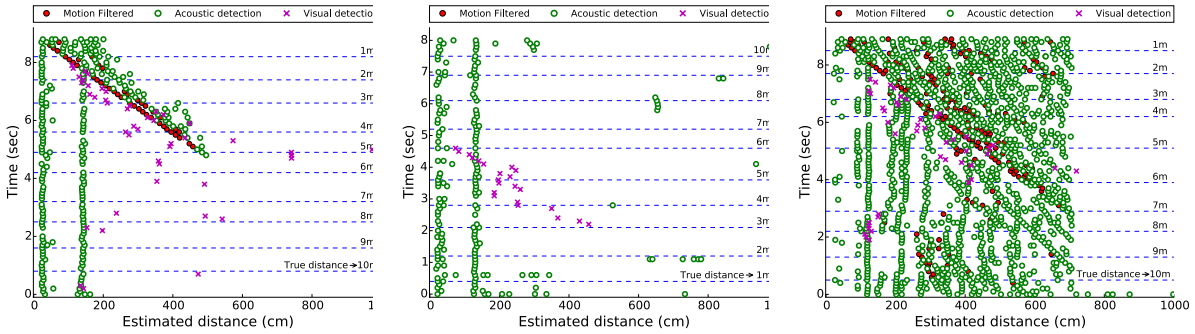


Figure 2.2: Example measurement of acoustic detection. Peaks of signals passing the matched filter indicate the reception of reflections from objects. The first and strongest peak represents the sound directly transmitted from phone speakers.

To identify the signals reflected from objects, the recorded signal is first passed through a bandpass FIR (Finite Impulse Response) filter and then through the corresponding matched filter as shown in Algorithm 1. At the n-th record, the highest-amplitude samples within a moving window are marked as peaks, P_n, if the signal's amplitude exceeds a threshold, thr. Due to the automatic gain control (AGC) in microphones and the different levels of environmental noise, thr is adjusted to the received noise level. The noise is observed from 600 samples before the sent signal is received, with the threshold set to α (mean($noise$)+std($noise$)) where α is set to 4 in `BumpAlert`. The width of the moving window, win_{peak}, is set to 40 samples, which is equal to the number of samples in the transmitted signal. The maximum resolution that can be discerned with these chosen parameters is about 15cm, which is equal to the product of the signal's duration and the speed of sound, so objects within 15cm of each other will be classified as a single object. Fig. 2.2 shows an instance of acoustic detection. The first peak indicates the sound sent out of the speaker, while the second peak is the reflection from the human body 28cm away, and the third peak is the reflection from the floor 142cm below the speaker. According to the ground truth, the error is less than 5cm in this case.



(a) Walking towards a wall (b) Walking away from a wall (c) Walking towards a wall in an aisle

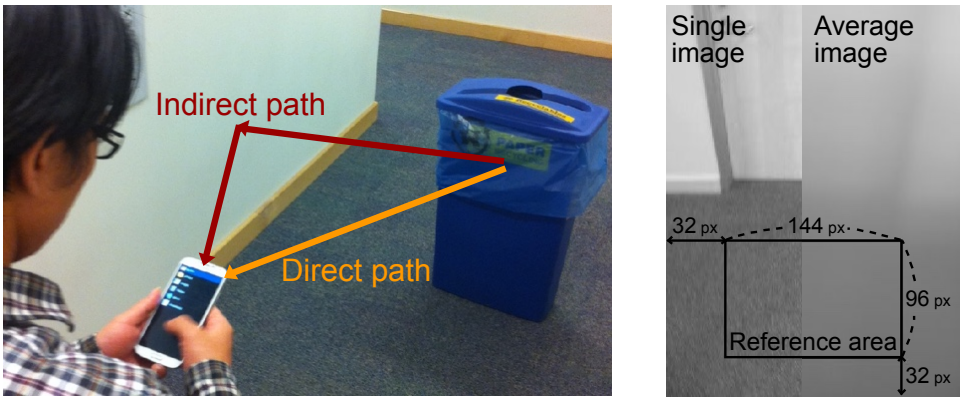
Figure 2.3: Distances to objects estimated by acoustic/visual detectors when a user walks toward/from a wall. A user is guided to walk towards (away) a wall from a location 10m away (1m behind). The circles and crosses represent the estimated distances to objects (including the wall) detected by our acoustic/visual detectors. The dotted lines represent the real distance to the target wall, which is collected via timestamped traces when users walk through pre-installed tags on the ground.

The signal used should be lower than a half of the sampling frequency for its accurate recovery. Ideally, a higher frequency is preferred because the sound of such a frequency will be less audible (hence less annoying) to the user, but the sent and reflected signals also degrade more at higher frequencies. On the other hand, decreasing the signal frequency will increase the time necessary to send a sufficient number of periods of the signal, which will lower the detection resolution. Note that there is no need to use a lower frequency signal. A lower frequency signal might incur less decay during its propagation, and can thus receive the reflections from farther-away objects. However, it also increases the time to wait for all reflections before sending the next sensing signal. There are also more environmental noises in the lower frequency band. According to our experimental results, the signal frequency of 11025Hz suffices to capture reflections within 2–4m, and reflections from objects more than 10m away are too weak to be detected for most devices we tested. This is what `BumpAlert` needs, enabling detection of nearby obstacles, and ensuring that all significant reflections are received within 100ms. Note that the current design of `BumpAlert` does not cope with the interference caused by multiple nearby devices. However, this problem can be avoided by utilizing existing multiple wireless access protocols.

For example, different devices can emit different frequencies of sound (FDMA) or different kinds of sound (CDMA) to ensure the emitted signals to have minimal correlation with each other. Testing such advanced settings is part of our future work.

The distance between the user and each object is computed as 1/2 the traveling time of the signal reflected from the object$\times$the speed of sound (331m/s). The performance of this scheme depends strongly on the ability to accurately record the time when signals are sent and their reflections are received. Errors of a few milliseconds will cause an estimation error of several meters due to the high speed of sound. Thus, any error between timestamps caused by non-real-time phone operating systems is unacceptable. We circumvent this problem by recording the time when the (reflected) signal is sent (received) and then computing the time difference between the sent and the reflected signals in terms of the number of consecutive samples [100]. As shown in Fig. 2.2, the signal identified with the largest magnitude, $peakMax$, is regarded as the sent signal, i.e., the signal directly going to the microphone. In Algorithm 1, this is used as the reference, $peakOffset$, for computing the time difference between the sent and the reflected signals. As the detection results shown in Fig. 2.3(a), when a user walks toward a wall (obstacle) from a 10m-away position, our acoustic detector is able to identify the reflection (as the diagonal green hollow circles) from the wall when the users are 5m away (as marked with the dotted line) at time 5. In this figure, the constantly appearing objects (two prominent vertical green hollow circles) estimated to be 30cm and 150cm away are, respectively, the human body and the floor.

One limitation of acoustic detection is that phone speakers and microphones are *omni-directional*, and hence the direction of the obstacle cannot be resolved. Another related problem is reception of multi-path reflections. The signal received by a microphone is actually a combination of the sent signal and multiple reflections of the same signal. Thus, an object actually 50cm away may cause a false detection as 150cm away due to multi-path reflections as shown in Fig. 2.4(a). This effect is severe, especially in an indoor environment where objects like walls and pillars cause multi-path reflections.



(a) Example posture and multipath reflections (b) Average of taken images

Figure 2.4: Assumed holding posture and its effect on detections. A wrong acoustic detection with a longer estimated distance happens due to multipath reflections. The marked area of images taken in this posture includes the ground texture with a high probability.

However, these two problems are greatly reduced by the `BumpAlert`'s need to detect only the closest object, i.e., the shortest-path reflection. Most reflections from objects behind the user are also absorbed/blocked by the user's body which is akin to the property of WiFi signals being blocked by the mobile users [113,133]. As shown in Fig. 2.3(b) where the user is walking away from a wall, the acoustic detector doesn't detect any prominent peaks of reflections even if the wall is just 1m behind the user at time 0. This feature helps the acoustic detector prevent from making wrong estimations when the object is actually behind the user. However, the detection results will still be affected by reflections from the side objects, such as walls and pillars. As shown in Fig. 2.3(c), the same experiment of a user walking toward a wall from 10m away is repeated but in a narrow (5m-wide) aisle. False detections are made due to side walls (vertical green hollow circles within a 2 – 6m range), making it difficult for `BumpAlert` to identify the real obstacle, i.e., the wall in front of the user.

To improve the detection results further, we introduce a *motion filter* that eliminates the detected reflections with 0 relative speed to the user. This filter is inspired by the results shown in Fig. 2.3, where all detected objects showing a constant distance to the users over time (*vertically* aligned circles) are unnecessary for the functionality of `BumpAlert` since

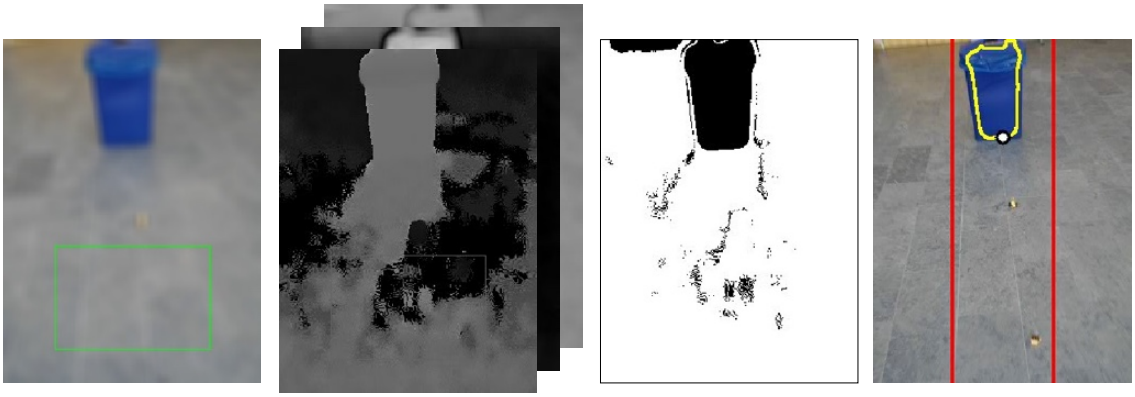
Algorithm 2 : Motion Filter

Input: detection distance of the n-th detection: D_n,
previous δ detections $D_{n-1}, \ldots, D_{n-\delta}$, depth: δ, walking speed: w
Output: results passing motion filter: D_{succ}

$D_{succ} \leftarrow \phi$
for $d \in D_n$ **do**
  $r \leftarrow 0$
  for i from 1 to δ **do**
    $d_{est} \leftarrow d + i \times w \times period_{detection}$
    for $d_{history} \in D_{n-i}$ **do**
      if $|d_{est} - d_{history}| < win_{error}$ **then**
        $r \leftarrow r + 1/\delta$
  if $r > r_{succ}$ **then**
    $D_{succ} \leftarrow D \cup d$
return D_{succ}

it is impossible for the users to bump into those objects without any relative speed to them. The user's walking speed is estimated by the phone's accelerometer as described later. The high-level goal of this motion filter is to remove detections with relative speeds unmatched with the user's walking speed. Thus, given the user's walking speed, w, and a history of δ previous detection results, $D_{n-1} \sim D_{n-\delta}$, only those reflections from objects moving at similar walking speeds are classified as the true obstacles toward which the user is walking.

As shown in Algorithm 2, the current detection, $d \in D_n$, is projected backward based on the user's walking speed, $i \times w \times period_{detection}$, yielding d_{est}. This is compared with the previously detected position of the object, $d_{history}$, and the probability of the presence of an object increases if the history matches the projection, i.e., $|d_{est} - d_{history}| < wind_{error}$. Yielding a probability, r, higher than a given ratio, r_{succ}, is said to *pass* the motion filter and identified as a positive detection. With this additional filtering, the reflections caused by objects without any matching relative speed, such as floor or side walls, can be filtered out as shown in Fig. 2.3(a). In this figure, detections passing a motion filter (marked as red solid circles) represent only the signals from target obstacles (i.e., the wall users walking towards) while the detections caused by human body, floor, and multi-path reflections inside the wall are excluded. A similar effect can also be found in Fig. 2.3(c), where most



(a) Blurred image (b) HSV transformation (c) Backprojection (d) Detection result

Figure 2.5: Visual detection. Images take by phone rear cameras are smoothed, transferred to HSV color scheme, back projected, and then filtered out blob detections.

reflections from side walls are also filtered out. However, the noisy detections in a cluttered environment cannot be completely eliminated by the motion filter. As shown in the same figure, more than 10 false detections caused by side objects pass our motion filter since those objects are too close to each other, resulting in a significant number of false positives which might annoy users. These false positives are reduced/removed by using the visual detector to ensure the detected object being in front of the user.

2.3.2 Visual Detector

To overcome the inherent limitation of acoustic detection, an additional sensing layer is added using the phone's rear camera. This removes the false positives and provides information of the object's direction. In `BumpAlert`, we assume that users will hold their phones in a typical position as shown in Fig. 2.4, and the rear camera's line of sight will be clear to capture objects in front of the user. `BumpAlert` can send the users texts or generate vibrations so as to maintain their phone tilt in its operational range. We have conducted a detailed survey of users' willingness to maintain their average phone tilt required for the functionality of `BumpAlert`. See the details of this users study in Sections 2.5 and 2.6.

There are two main challenges to detect objects in the rear camera view. The first is to determine the presence of objects and the second is to determine the distance between

the user and the objects due to the lack of depth perception in the images taken by only a single camera. `BumpAlert` does not use any *a priori* information, such as the shape and color, to identify the presence of objects. Having no prior knowledge makes `BumpAlert` more general, enabling detection of any type of dangerous objects and preventing collision with them. Detecting objects without any prior knowledge is difficult, though. The goal of `BumpAlert` is, however, not to identify every object in the scene but to know if there is *any* nearby object in front of the user. Specifically, `BumpAlert` adopts the back-projection technique in [48, 121] to identify the objects that are different from the ground/floor. Its idea is to use the texture of the ground surface where the user is walking on and to compare it with the rest of the image, looking for textural similarities and dissimilarities. As shown in Fig. 2.5, a 10×10 blurring filter is used first to reduce the noise from the image, and the image is then transformed into the HSV space. The back-projection algorithm is applied to determine which parts of the image are not related to the ground/floor texture. The last step is to apply an erosion filter to remove any residual error from the back-projection algorithm. After completing these steps, blobs with areas larger than a predefined threshold are identified as obstacles and the point closest to the bottom of the image is returned as the closest obstacle.

Some astute readers might observe that the key assumption in the back projection is the knowledge of ground/floor texture. In case an object is erroneously included in the region as a reference of ground/floor, the object won't be seen by our visual detector because the back projection classifies the object as a part of ground/floor. Identifying the ground/floor in an arbitrary image is difficult, but does not cause problems to `BumpAlert`. Images are only taken when users are walking and using their phones at an assumed position as shown in Fig. 2.4. Under this assumption, we can ensure that a specific area in the image can represent the information of the ground/floor with a high probability. In order to determine this area, we conducted an experiment with 10 participants. They were requested to take pictures while using their phones at a comfortable position 2m away from a door. The

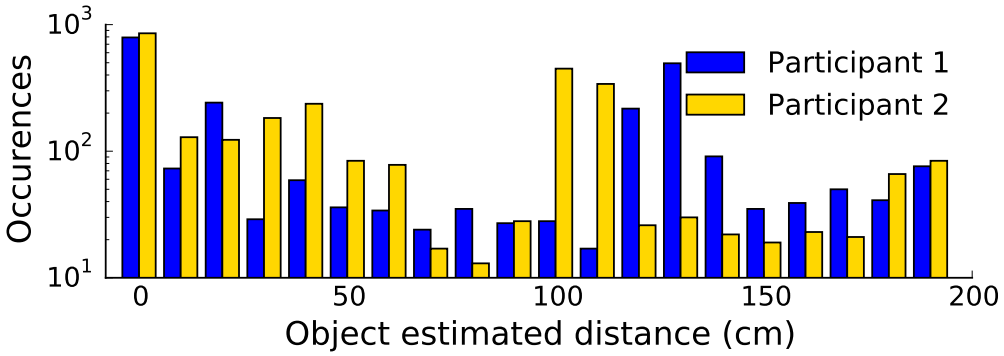


Figure 2.6: Height estimation by the acoustic detector. The phone's height can be estimated by the sound reflections from the ground.

average of all the pictures taken is shown in Fig. 2.4(b), where the dark area indicates the area consisting of ground/floor. The size of that area we choose is 96 pixels$\times$144 pixels located 32 pixels above the bottom of a 240$\times$320 image. The area is chosen above the bottom of the image since it is possible to include the user's feet in the bottom area.

After the closest point of objects in the image is identified, a pixel difference from the detection point to the bottom of the image is defined as p, and the pixel-to-real-world distance transform is computed as $d = \text{pixel\_to\_distance}(p, h_p, t_p)$, where d is the real-world distance to the detected object, h_p and t_p represent the height and the tilt of the user's phone with respect to the ground. A detailed derivation of this transform based on a camera projection model can be found in [53]. This computation is possible only if the height and tilt of the phone are known. As these two parameters are not fixed when people are walking, a method is needed to estimate them online. The phone's tilt can be directly acquired from the accelerometers as $t_p = cos^{-1}(acc_z/acc_{mag})$, where acc_z is the acceleration orthogonal to the phone's surface and acc_{mag} is the magnitude of overall acceleration caused by the user's motion or the earth gravity. In contrast to derivation of the tilt from the accelerometer readings, the phone's height is unknown when the user is walking. *`BumpAlert` utilizes the results of the acoustic detector to estimate the phone's height.* This design is novel since existing image-based detections simply assume the height of a camera is known. This parameter might be easy to acquire in certain scenarios, such as installing the camera at a

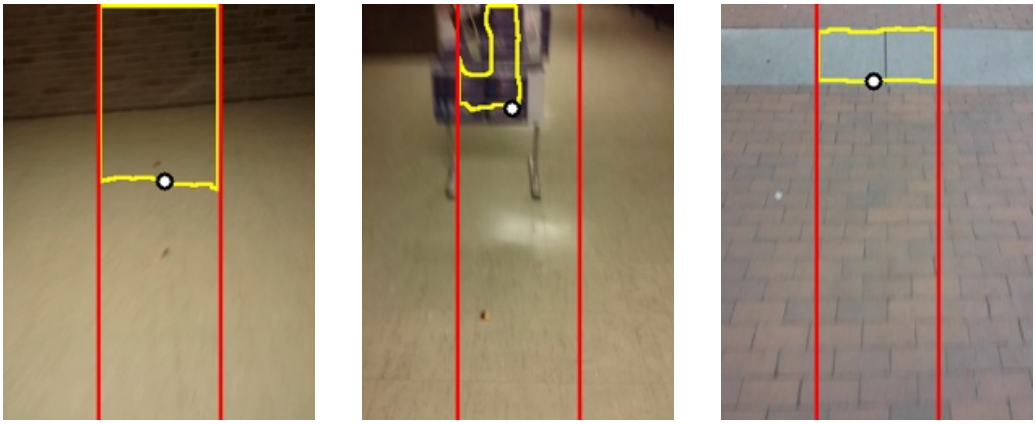
fixed location inside a car, but not in our scenario, since the height of a phone vary with users depending on their height and ways to hold the phone.

The histogram of objects detected by the acoustic detector with different estimated distances are plotted in Fig. 2.6. This data were collected for two participants of different heights. The maximum peak at distance 0 is the receipt of the transmitted signal. Detections within [10, 60]cm are reflections from the human body. There are also relatively fewer detections in the region [70, 180]cm. The main reason for this phenomenon is that people need a space in front to move forward, resulting in a low probability that there is an object in this range while people are walking. Thus, the highest peaks within this range are actually the reflections off from the floor. As shown in the figure, this is approximately 120 to 140cm for participant 1 and ranges from 100 to 120cm for participant 2. By tracking the distance in this range, we can estimate the phone's height with an error of less than 20cm.

Although the visual detector can determine both the direction and distance of objects in front of the users, it is not desirable for constant/frequent use for the following reasons:

- computational cost of image processing is much higher than acoustic detection, thus consuming more battery;
- distance measured is less accurate than acoustic detection due to the changing tilt and height estimations;
- back projection may be inaccurate for complex floor patterns; and
- falsely identifying pattern transitions on the ground/floor as obstacles.

From our experiments, we found the false positives of visual detection caused by the following three factors as shown in Fig. 2.7. First, shadows cast on the ground will cause the color of the ground/floor to be different from its surroundings, hence flagging as a different texture area. Second, overhanging obstacles cause the estimated distance to be farther away than the actual position because their bodies are not fully connected to the floor. Third, changing patterns of the ground/floor also cause false detections and are mistaken as an obstacle as they are different from the identified ground/floor texture. A representative



(a) Shadow (b) Overhang (c) Patterns

Figure 2.7: False detection by the visual detector. The visual detector might over/under-estimate the distance to objects in different scenarios.

error pattern of visual detection can also be found as the purple crosses shown in Fig. 2.3. For example, there is a burst of false positives between 3 and 5 seconds in Fig. 2.3(b), even though there weren't any objects ahead. The detection is also less accurate than the acoustic detector. As shown in Fig. 2.3(a), the estimation errors of the visual detectors between 5 and 8 seconds are about 10–100cm while the acoustic detector has errors less than 15cm. `BumpAlert` overcame the above challenges by *combining* the acoustic and visual detectors as described in Section 2.3.4.

2.3.3 Motion Estimator

As mentioned in the previous section, the tilt of a phone's camera is directly related to its accelerometer. Similarly, the acoustic detector needs feedback from the phones' sensors that provide information about the user's walking speed to improve the detection accuracy. Using the accelerometer readings, the steps that a person takes can be detected as there exist periods of high and low accelerations. Each peak-to-peak cycle indicates if a step has been taken and the walking speed can be estimated as the product of step frequency and average step size. In `BumpAlert`, the step size can be either entered by the user or set to the default average step size. This coarse estimation of walking speed is adopted in various applications, such as dead-reckoning systems [137]. The acceleration can also

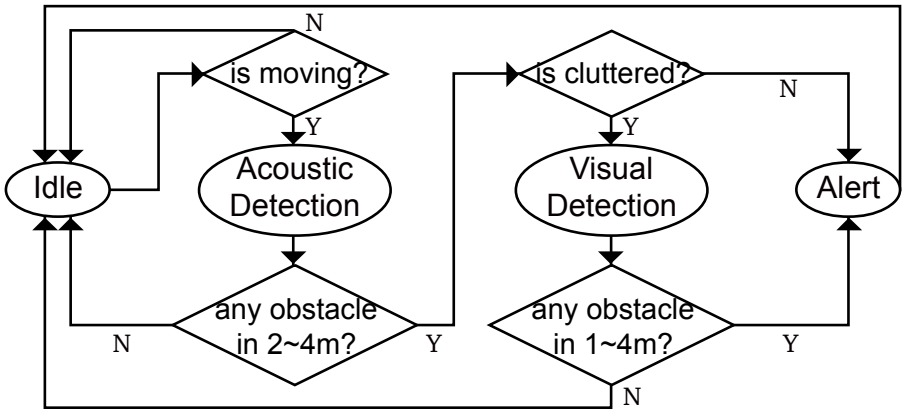
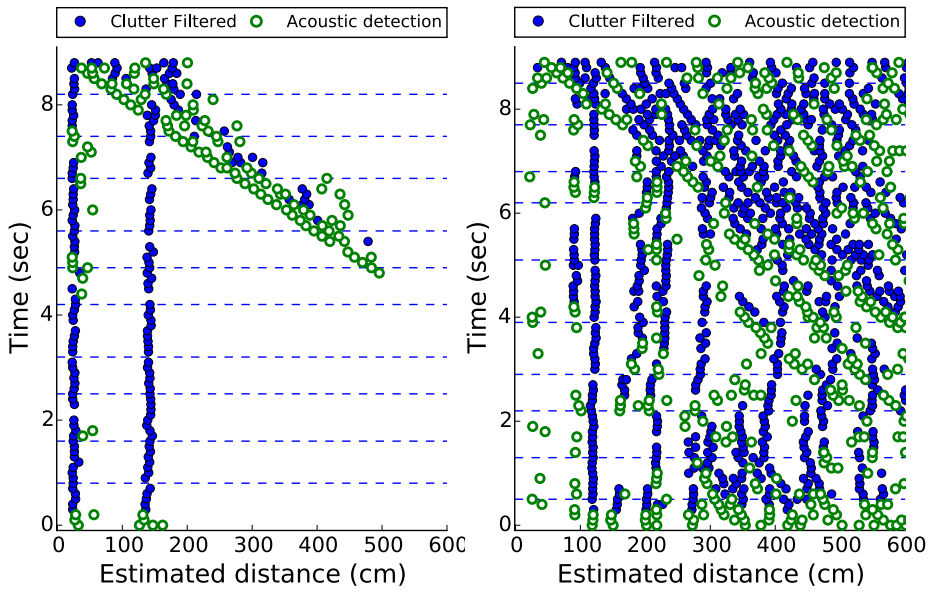


Figure 2.8: Fusion algorithm. If necessary, the visual detector can be enabled to check if the objects found by the acoustic detector do indeed exist.

allow the system to determine if the user is walking or stationary when its variance exceeds a predefined threshold.

2.3.4 Fusion Algorithm

A combination of the above algorithms is used to improve accuracy and lower the false detection rate. We also reduce power consumption by deactivating components that would not improve the detection accuracy. Fig. 2.8 shows the logical flow of when to run which component based on outputs from other components. First, the detection algorithm need not be run when the user is stationary. We trigger the the detection algorithm only when the user is walking, and switch it off when there is no movement. Second, the low-cost acoustic detector is triggered before the high-cost visual detector. That is, the visual detector is triggered to double-check the acoustic detection result only when the latter is not convincing enough. When the visual detector is enabled, a warning message is issued only when the both detectors find the same object within a 2–4m range. The acoustic detector is good at detecting the objects around the user within a certain range but is less effective in dealing with side objects (i.e., in cluttered environments). In contrast, the visual detector is free from the side object problem since it focuses on the user's front view. `BumpAlert` therefore uses a combination of acoustic and visual detections, especially in cluttered environments.



(a) Outdoor (b) Aisle

Figure 2.9: Objects identified by the clutter filter. The clutter filter is a special case of the proposed motion filter for finding objects with 0 relative speed to users. It provides a hint for the fusion algorithm to trigger the visual detector, when necessary.

To identify cluttered environments, the motion filter in Algorithm 2 is also used to estimate the number of stationary objects when we set the relative speed to 0. This new application of the motion filter with 0 relative speed is called the *clutter filter*, and its effectiveness is shown in Fig. 2.9. It can detect those objects that do not have any relative speed to the user. The outdoor environment does not leave many objects after applying the clutter filter, while the aisle environment leaves many objects. Thus, the aisle environment can be identified as cluttered since the number of objects passing the filter exceeds a predefined threshold. Reuse of the motion filter for identifying a cluttered environment is also a novelty of `BumpAlert`, which provides a hint to the fusion algorithm for triggering the visual detector. Existing approaches based on light and geomagnetism changes can only determine if users are located inside a building or not [135], which is not sufficient since disabling the visual detector in a lobby area (indoor) is found to provide better results, but the visual detector is necessary in a cluttered (indoor) aisle.

In the fusion algorithm, different detectors complement each other in different situa-

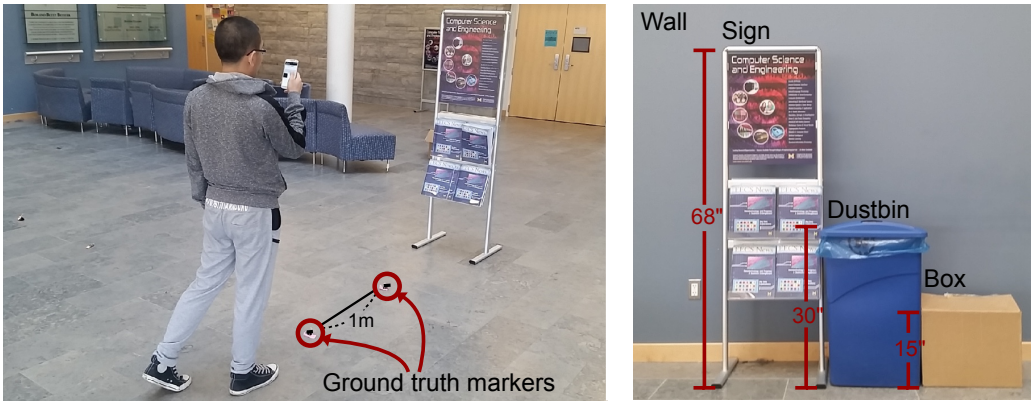
tions. In a cluttered aisle, side walls will be falsely classified by the acoustic detector as obstacles, but are filtered out by the visual detector since such side objects are not captured by the rear camera. On the other hand, crossing from a cement floor to a grassy area is falsely classified as obstacles by the visual detector but is filtered out by the acoustic detector because no reflections are received from the grassy area. By integrating these detectors, `BumpAlert` can therefore discover dangerous objects with high accuracy and a low false positive rate. Note that our current design aims to prevent users from bumping into static objects, like walls, signboards, or pillars. See Section 2.7 for the discussion of detecting moving objects.

2.4 Implementation

We implemented `BumpAlert` as an Android app on the Galaxy S4. As `BumpAlert` relies only on the phone's built-in sensors, it can be easily ported to different platforms, such as iOS and Windows. For `BumpAlert` to be computationally efficient, the signal processing, such as bandpass and matched filters, are implemented in C and interfaced through the Java Native Interface (JNI), which yielded shorter execution times. The control logics shown in Fig. 2.8 are implemented in Java due to its low computation requirement. As a result, each iteration of the acoustic/visual detector can be completed within 25/80ms while its period is set to 100ms.

We choose the rate to trigger acoustic/visual detectors to be 10Hz and the sensing range to be 2–4m in order to balance between detection accuracy and processing cost. According to the results in [41,42], the average human walking speed is about 1.5m/s and the reaction time to an auditory alert is about 150ms. This reaction time is similar to using a vibration alert. Thus, a sensing period of 100ms with a distance range of 2–4m is sufficient to alert the user, and the choice of these parameters works well as shown in Section 2.5.

To run `BumpAlert` with other apps simultaneously, we may choose to implement `BumpAlert` as a system service. However, in the latest version of the Android API, the camera is not



(a) Test environment (b) Target obstacles

Figure 2.10: Test setting. Ground truth markers are used to collect the real distances to the test targets. The selected test targets are ordered by their size, which is related to detection accuracy.

allowed to be used in a background service due to privacy issues. Likewise, in `BumpAlert`, images are not saved but only processed for object detection. In future, we will implement `BumpAlert` as an open-source library so that app developers may easily include our modules to enable this functionality to protect their users.

2.5 Experimental Evaluation

We have conducted a series of experiments to assess the performance of `BumpAlert` in real-world settings. Since the goal of these experiments is to capture and evaluate the performance of `BumpAlert`, we manually selected objects of different sizes and asked participants to walk toward those objects multiple times under different representative scenarios. The benefit of this setting is to collect ground truth and quantize the detection of `BumpAlert` accurately. This information is important for us to infer the performance of `BumpAlert` in the real world but difficult to obtain if participants are allowed to walk toward random obstacles in a single long route. Moreover, as shown in our experiments, the performance of `BumpAlert` depends on the objects and scenarios, so the objects seen in a long route create a significant bias in the final result. For example, a path consisting of 10 walls and 5 dustbins can get a better result than the one of 5 walls and 10 dustbins because

wall is an easy target to detect. To avoid this bias, we chose to provide the accuracy of `BumpAlert` against each object in different scenarios rather than the aggregated accuracy in a single long route. The usability of `BumpAlert` is evaluated further in Section 2.6 via a users study that collects feedbacks from 21 participants who used `BumpAlert` for 15min. In future, we plan to evaluate `BumpAlert` with more participants over a longer period of time after its deployment.

In each experiment, 7 participants are instructed to walk towards various objects, such as walls and signboards in both indoor and outdoor environments. Each of these experiments is repeated 10 times to average the errors due to the differences in each user's walking pattern and path. The participants are instructed to press a specified button when they walk through a marker placed on the ground as shown in Fig. 2.10(a). This serves two purposes. First, it simulates the users being pre-occupied with a task that they would have to accomplish by looking at, or typing on their phones. Second, the ground truth can be collected as the markers are placed at a 1m interval. In this evaluation, we define a positive detection as the obstacle detected within a 2–4m range. Any alert when the user is 2–4m away from the target object is classified as a successful alert and the ratio of these alerts is called the *true positive* (TP) rate. On the other hand, any alert occurring when the user is actually 4m or farther away from the target object is classified as a false alert, and the corresponding rate is calculated as the *false positive* (FP) rate. The average delay is defined as the time from a participant walking through the 4m marker to the time an alert is triggered.

2.5.1 Accuracy in Different Environments

In this experiment, a set of 4 objects shown in Fig. 2.10(b) are used as obstacles in 3 different environments. They are wall, signboard, dustbin and cardboard box, which are ordered by their relative size. These objects are selected to represent different types of objects in the real world. The difficulty of detection is due mainly to the size of objects. For example, we get similar results for the detection of a glass door and a wall. Moreover, since

	Wall			Signboard			Bin			Box		
	TP (%)	FP (%)	Delay (ms)	TP (%)	FP (%)	Delay (ms)	TP (%)	FP (%)	Delay (ms)	TP (%)	FP (%)	Delay (ms)
Out./Acoustic	100	0.6	320	98.3	5.6	516	96.7	2.8	567	91.7	3.5	470
Out./Visual	63.3	9.8	247	85.0	27.6	265	85.0	13.9	251	75.0	19.9	485
Out./Fusion	100	0.5	433	98.3	2.2	610	95.0	1.7	572	90.0	1.8	508
Lob./Acoustic	98.3	1.3	108	93.3	1.2	318	96.7	0.6	278	93.3	1.3	321
Lob./Visual	78.3	11.9	297	61.7	4.5	323	86.7	12.7	496	71.7	7.5	711
Lob./Fusion	98.3	1.3	111	93.3	1.0	367	96.7	0.6	290	93.3	1.3	325
Ais./Acoustic	100	32.1	105	100	28.2	193	100	29.1	203	98.3	28.0	245
Ais./Visual	98.3	28.4	45	100	27.5	598	100	22.2	417	100	26.8	465
Ais./Fusion	91.7	9.8	330	100	6.0	547	95.0	6.1	447	91.7	6.3	566

Table 2.1: Comparison of performance in different environments

these objects can easily be found/moved in both indoor and outdoor environments, the performance degradation caused by different environments can be accurately measured in this setting. Other objects, such as pillars and cars, are also tested and shown to have similar characteristics but those results are omitted due to space limitation. The three test environments we used are an open outdoor area, a building lobby, and a (5m–wide) cluttered aisle. Each participant repeats each experiment 10 times and the 10Hz raw data of both acoustic and visual detectors are logged to evaluate the detection rate of individual experiment offline by the same detection program. This is to allow for comparison of each individual component based on the same data set, which consists of more than 12km walking traces. We conducted experiments in the presence of environmental noises, such as students' chatting, but found those noises didn't affect `BumpAlert`'s performance much since the frequencies of most noises associated with human activities are below 11kHz [116] and `BumpAlert` adjusts its detection based on the noise level. The only problem we found is participant 7's outdoor trace collected on a very windy day (more than 24mph). In this case, the signal received at the phone's microphone was saturated by the wind sound alone, and hence, we postponed the experiment to the next day.

From the results in Table 2.1, one can see that the acoustic detector outperforms the visual detector in TP rate because the sensing range and sensitivity of the former is longer

and better than the latter. The overall TP rate of acoustic detection is higher than 95% which is sufficient to identify most dangerous objects. The average delay in all cases is shorter than 650ms for both visual and acoustic detections. This low delay of `BumpAlert` provides the users walking at 1.5m/s with more than 2s to react and avoid the obstacles, which is much longer than the human's reaction time [41].

The aisle scenario shows a high FP rate for the acoustic detection due to its cluttered environment. In contrast, the visual detection is not affected by this scenario due to the directional nature of image taken by the phone's rear camera. Therefore, the average FP rate of visual detection in this scenario is even lower than the FP rate of acoustic detection. We exploit this complementary nature of acoustic and visual detectors by using a fusion algorithm to ensure a high TP rate in outdoor environments while significantly reducing the FP rate in indoor environments as shown in Table 2.1. The fusion algorithm also lowers the FP rate in outdoor environments which are due mainly to a strong wind blowing into the phone's microphones. Actually, many *false* detections in the 5m-wide aisle are not incorrect since there exist objects, e.g., the water fountain on the side and the emergency sign at the ceiling, which is *in front of* users within 2–4m. If the detection range is shrunk to 1–2m, the FP rate of acoustic detector is reduced from 28% to 5% and it is reduced further to 2% when a combination of acoustic and visual detection is applied. Note that there is a trade-off between false detection and detection range. One possible resolution is to alarm users when they are located in a cluttered environment where the detection range of `BumpAlert` is shrunk, and hence they need to pay closer attention to their walking. This is part of our future work.

As stated in Section 2.3, the key component of the fusion algorithm to work properly is its ability to estimate the number of stationary objects through the clutter filter. In real-world experiments, we set the threshold of stationary objects to classify environments as cluttered as 5 (i.e., turning off the visual detection when there are < 5 stationary objects). The distribution of stationary objects in different scenarios is plotted in Fig. 2.11. Our

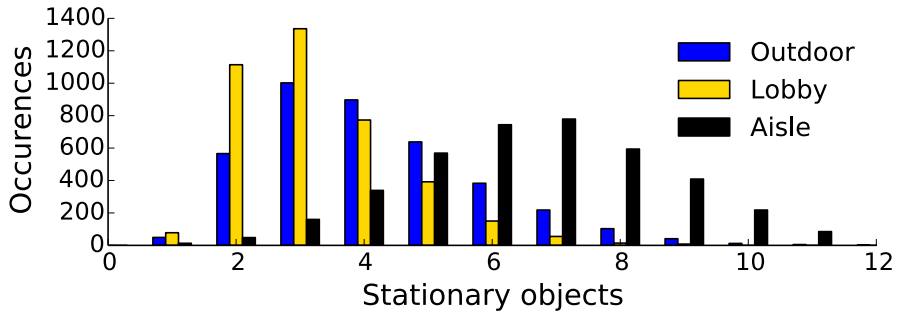


Figure 2.11: Stationary objects passing the clutter filter. Cluttered areas can be identified by monitoring the number of stationary objects.

experimental results also validate the effectiveness of the clutter filter in enabling the visual detector under a proper condition.

Of all the objects considered, the wall is found the easiest to detect due to its large size, and the box is the hardest in terms of TP ratio and delay. Moreover, the TP rate of signboard detected by the visual detector is lower than that of other objects, which is due to the signboard overhanging above the floor as shown in Fig. 2.7(b). Although the visual detection for the signboard is above 80% in outdoor and aisle environments, their high TP rates are also accompanied by a high FP rate. This implies that the detector was guessing most of the time, leading to the high TP and FP rates and not a true representation of accurately detecting the object.

Many other objects have also been tested but the results are not reported here due to space limit. One interesting finding is that acoustic detection of a human is harder than a box even when the human is much larger than the box. This is because the human body absorbs most sound signal instead of reflecting it. We found that the acoustic detector can only detect humans within a 1–3m range under the current setting, which is shorter than the other objects we tested. Nevertheless, `BumpAlert` can still detect humans with a TP ratio higher than 82%. Moreover, the chance of bumping into a person is less likely than other stationary objects because people usually try to avoid distracted walkers. An alternative solution to this problem is to continuously monitor objects with an additional signal of different (low) frequency which is easy to be reflected by the human body.

id	acoustic TP(%)	acoustic FP(%)	visual TP(%)	visual FP(%)	$\bar{h_p}$ (m)	$\bar{t_p}$ (°)
p1	97.5	5.4	97.5	36.4	1.3	52
p2	100.0	1.8	100.0	11.1	1.1	54
p3	95.0	2.5	87.5	21.3	1.3	53
p4	100.0	3.2	90.0	17.6	1.1	39
p5	90.0	0.2	12.5	2.7	1.0	31
p6	100.0	1.7	100.0	32.2	1.2	65
p7	100.0	2.6	100.0	17.6	1.2	56

Table 2.2: Individual detection rate of the trace in lobby

Even though the current version of `BumpAlert` doesn't handle moving objects, it is general enough to detect a variety of objects in real time. The issue of detecting moving objects like humans or cars will be part of our future work, and it might be addressable by using other complementary approaches such as those in [65, 126].

2.5.2 Accuracy among Different Participants

To study the effects of different participants with different phone-holding positions and walking patterns, the above results are separated based on individual participants. The phone tilts/heights and the corresponding detection results are summarized in Table 2.2. According to our experiments, the tilt of phones, t_p, varies from 31° to 65° among different participants; so does the phone height h_p vary from 1 to 1.3m. These parameters for the same user did not vary much over time.

An interesting finding is that the acoustic detection accuracy is slightly different among participants. We have repeated several tests with different holding positions and found that the variation is affected by the way the phone is held and the AGC of the phones' microphones. For example, when the speakers are being blocked by fingers, the received signal strength is low due to the obstruction. On the other hand, if the phone is held tightly, the magnitude of the received signal sent directly from the phone is increased. This signal may be strong enough to saturate the range of the microphones, and the reflected signals

are usually weaker due to the lower gain adapted by AGC. However, with the adaptive threshold mechanism as described in Section 2.3, `BumpAlert` can accurately estimate the noise level and detect reflections effectively.

The extreme low visual detection ratio of participant p5 was caused by his way of holding the phone, 30° with respect to the horizontal plane. The detection results we collected from participant p5 show that only those images close to (within 1m of) the obstacles can yield a sufficient area for detection because of the low tilt of the phone, implying that our visual detection is not applicable to certain postures of holding a phone. We also recruited two additional participants who hold their phones with a posture similar to participant p5's to repeat the above experiments. Our results indicate that the visual detector is unable to function with tilt lower than 30° for identifying 2m-away objects. However, the high probability of successful visual detection by the other users also implies that visual detection works with a broad range of tilts from 40° to 65°. One potential way to address this issue is to warn the users, when they enable `BumpAlert` but hold their phones with the tilt less than 40°. According to the users study in Section 2.6, most users feel comfortable with this operating range of `BumpAlert`.

2.5.3 Processing Cost and Energy Consumption

Our final experiment is to evaluate `BumpAlert` for its real-time performance and resource consumption. Under its four different configurations, we ran `BumpAlert` for an extended period of time in typical environments. The CPU usage of `BumpAlert` is logged via the *top* command at an interval of 10 seconds. A 1-hour trace is averaged to obtain CPU usage as well as power consumption. Four different scenarios are tested: idle (with backlight on), acoustic detection only, visual detection only, and trace. The idle case is used as a baseline which mainly represents the power consumed by the backlight. In the case of acoustic or visual detection only, each algorithm is run independently at 10Hz with backlight on. Since the energy consumption depends on how often `BumpAlert` turns on/off

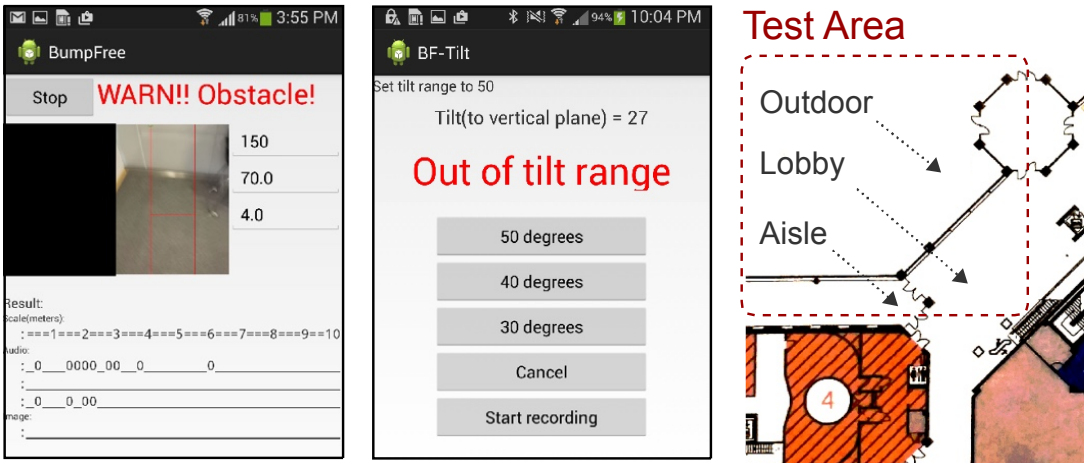
the visual detector, we also include a real-world trace from participant 1 where the visual detector was enabled only when necessary. This trace is collected when the participant is walking between his home and work. We chose to display participant 1's result because his on-foot travel time is longer than the other participants.

The CPU usages when the app is Idle, in Acoustic only, and Visual only are 3.08%, 8.92% and 17.80%, respectively. One can see that the CPU usage of Visual detector is approximately twice the value of Acoustic detector. As the high CPU usage, the power consumption of the visual detector is also observed to be much higher than the acoustic detector's. For example, the acoustic detector only consumes one-fourth more energy than the idle baseline (with backlight on) but the visual detector consumes twice more energy. In our experiments, most of the energy is consumed by the microphone/speaker/camera hardware, not by the computation [39]. Thus, the capability of reducing the energy consumption in software is limited. Note that the additional energy consumed by `BumpAlert` will be reduced further when users turn on WiFi/4G or play mobile games. In the actual usage as the trace of participant 1, the S4 battery only has an additional 8% drop after one hour usage.

2.6 Usability Study

We randomly selected 21 passers-by (10 females and 11 males) in our campus without prior knowledge of `BumpAlert` to evaluate its usefulness and practicality. The users were asked to try out `BumpAlert` for 15 minutes and fill out a survey form. Users tried a demo version of `BumpAlert` as shown in Fig. 2.12(a) at locations shown in Fig. 2.12(c). The results are summarized in Table 2.3.

The first section of our survey attempts to analyze the prevalence of distracted walking. Our result indicates that 81% of the participants use their phones while walking and 43% of them had run into obstacles due to distracted walking. Even though a half of the participants did not bump into any obstacle before, 76% of them were afraid of running into



(a) Demo app (b) Tilt survey app (c) Test location

Figure 2.12: Survey settings. The demo version of `BumpAlert` processes the acoustic/visual detectors in real time. The separate tilt survey app records phone tilt when participants walk and provide feedback when the phone tilt is not in the selected range.

Questions	Disagree	No opinion	Agree
I can play my phone around 40° (at walking for detecting obstacles)	10	0	90
I can play my phone around 50°	10	18	72
I can play my phone around 60°	80	5	15
Detection accuracy is helpful	14	14	72
Detection range is acceptable	28	0	72
False alarm is bothering	39	32	29

Table 2.3: Survey results (%).

obstacles when they use their phones while walking. The percentage of people colliding with obstacles increases to 86% if their friends who had bumped into objects are included.

The second section of the survey attempts to know the tilt when the users hold their phones and check if people are willing to hold phones in a specific tilt range for the benefit of obstacle detection and warning. A separate Android survey app shown in Fig. 2.12(b) was used to record and inform the participants of the tilt in holding their phones. They were first asked to walk with `BumpAlert` enabled to record tilts when they hold their phones in the most comfortable position. Then, we selected several different angles that allow the survey app to monitor the tilt of phones and provide a feedback (via vibration and a red text) when the user doesn't hold the phone in the selected angle within a $\pm 10^\circ$ range.

The phone tilt has been studied extensively in [111] by continuously recording the tilt via published Android widgets. However, the users' state (e.g., walking or sitting) when the tilt is recorded was not reported there. In our users study which records the phone tilt when users are walking, most participants hold their phones at approximately 35° relative to the ground. This result matches the average phone tilt when Google Maps is run as reported [111]. This tilt distribution is not optimal for `BumpAlert` as shown in Section 2.5. However, after having experience in holding phones with different angles and being told about our purpose, 90% of participants were willing to hold their phones between 40° and 50°, which is proven good for `BumpAlert`. Thus, it is reasonable to provide similar feedback when `BumpAlert` is enabled but the tilt of phones is not in the operation range.

The third section of the survey asks participants to evaluate the usefulness of `BumpAlert` after a 15min trial in three scenarios as shown in Fig. 2.12(c). The three criteria we used are the detection accuracy, detection range and false-alarm rate. About 72% of the participants agree that the detection accuracy and range are adequate, allowing them enough time to react to imminent obstacles. Some participants have commented that they would be able to avoid obstacles at even a shorter distance, such as 1.5–3m. This feedback was useful for `BumpAlert` to reduce the false positive ratio. 29% of the participants did not want to have any false alarm. We found some of participants react even to the correct detection of a wall 4m away as a false alarm. Based on the performance of `BumpAlert` we were able to satisfy most participants with low false positive rate and good detection ratio.

The last section of the survey addresses the issue of power consumption. Only 14% of the participants want the power consumption to be below 4% per hour. The power consumption of `BumpAlert` varies from user to user, depending on the users' activities. In our initial experiment, power consumption is approximately 8% per hour, which meets the criteria of 86% of people who are willing to use the application.

Even though the study of 21 users is somewhat limited, it did help us understand what the users need. For example, besides the quantitative results mentioned earlier, during the

user study, we also noticed that the users' satisfaction with `BumpAlert` is strongly dependent on the user interface (UI). For example, in a crowded area, users are more comfortable when the UI shows a detailed notification like "*Crowded area detected. Don't use your phone while walking*" rather than a message like "*BumpAlert is off*". Many of the feedbacks we received actually made us adjust our design as shown in Section 2.8. Crafting a proper UI and building a large-scale user study are parts of our future work.

2.7 Limitations and Discussion

Based on our evaluation and users study, `BumpAlert` has been proven able to prevent distracted walkers from colliding with various obstacles, ranging from glass doors to small dustbins. However, there are a few limitations of the current version of `BumpAlert`, including the detection of moving objects, the liability of missed detections, and the audible sound used by the acoustic detector, as discussed below.

2.7.1 Detection of Moving Objects

In addition to the various static objects we have already tested for the evaluation of `BumpAlert`, its current version cannot detect moving objects since they have the unmatched relative speeds and are thus filtered out by the motion filter. There are several potential solutions to address this issue. For example, instead of just matching the pedestrian's walking speed with the speeds of objects moving toward the user, a more sophisticated machine learning algorithm might be able to distinguish the detections caused by different objects, and then track their moving trajectories. However, this type of complex algorithm might consume more energy/computation resources, and generate additional false detections. Finding a balance between detection capability and computation cost is part of our future work.

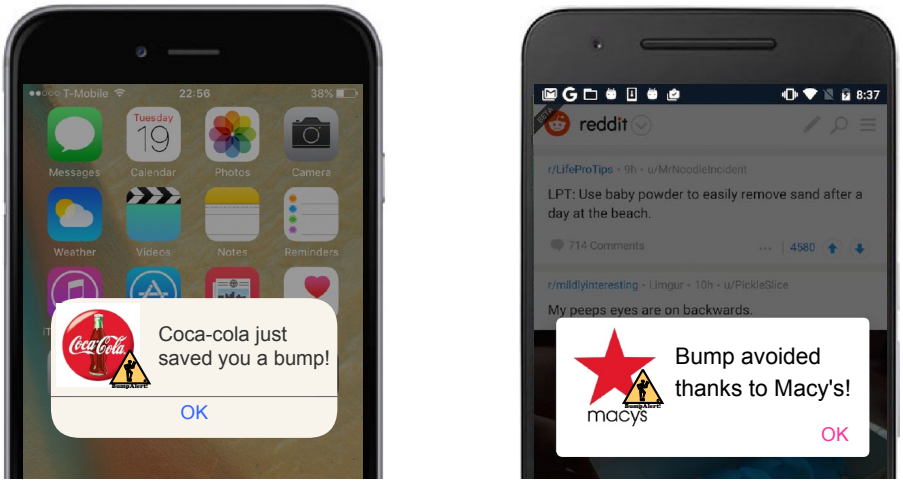


Figure 2.13: An example user interface for the business of app developers. `BumpAlert` executes in the background with no disturbance to users and the warning with third-party advertisements is shown only when dangerous obstacles are detected.

2.7.2 Liability of Missed Detections

As mentioned earlier, `BumpAlert` is unable to warn users of "all" dangerous objects, and it is also not the purpose of `BumpAlert`. Some objects might be detected by integrating `BumpAlert` with other existing systems while others may not. For example, distracted pedestrians might fall by stepping through the gap from sidewalks to streets, but `BumpAlert` won't be able to detect this gap since there is nothing in the gap to reflect the audio signals. This situation can be prevented by incorporating an existing system designed specifically for recognizing the street gaps [70]. The same principle can also be applied to the detection of moving vehicles [126]. However, no matter how the system is integrated and designed, there will always be possible missed detections. That is, all warning systems including `BumpAlert` are to enhance, but not to guarantee, distracted walkers' safety.

We argue that even an expensive system relying on many specialized sensors still experiences miss detections, e.g., the recent tragic accident of the latest Tesla autopilot driving model [30]. The main goal of `BumpAlert` is to provide distracted pedestrians additional safety protection with only minimal resources. So, users should not expect to navigate based solely on `BumpAlert` but exploit the `BumpAlert`-provided warning for their safety.

Questions	Disagree	No opinion	Agree
I can tolerate 11kHz sound beep (on the purpose to detect obstacles)	42	10	48
I can tolerate 4kHz sound beep	48	10	42
I can tolerate 441Hz sound beep	42	29	29
I can tolerate 11–22kHz chirp	95	0	5
I can tolerate Music fused beep	39	29	32

Table 2.4: Audible sound survey (%).

Fig. 2.13 shows an example user interface for developing `BumpAlert` as a freemium which lowers the users' expectation of 100% detection rate. App developers still get paid via advertisements in the alert view when obstacles are detected correctly.

2.7.3 Annoyance Caused By Audible Sound

As mentioned before, `BumpAlert` relies on a 11kHz beep to sense environments. Although only a short (i.e., 40 samples) sequence of sound is emitted, imperfect speaker design makes the beginning and end of this sound louder than expected. This audible noise is due mainly to the hardware limitation of commodity phones.

The authors of [81] have shown that a 22kHz sound can be used to send data at a low bit-rate with proper signal processing. However, the purpose of `BumpAlert` is different from theirs in that the emitted sound should be strong enough to generate reflections from obstacles rather than sending data in a best-effort manner. Moreover, their results didn't account for the limitation of speaker hardware either, since a special speaker (unavailable in commodity phones) was used in their evaluation. In our experiments with Galaxy S4/5, inaudible sound of 22kHz is unable to detect objects within 2–4m. This result is also consistent with their hardware study; the signals captured by certain commodity phones at 22kHz are 30dB weaker than those in the audible range. The responses to this audible sound among 21 participants are summarized in Table 2.4.

The participants were asked to answer the questions after trying `BumpAlert` and based on the assumption that it can help them avoid collision with obstacles during distracted

walking. As shown in this table, even with prior knowledge of BumpAlert's purpose, only 48% of them support the sound emitted by the current version of BumpAlert. Other lower frequency sounds received even less support. Use of a wide-band chirp, which can further enhance the accuracy via pulse compression, was rejected by 95% of the participants. An interesting candidate to *hide* the audible beep is to fuse the signals into a music. For example, an instrumental music is selected and the music signal of 10–12kHz is filtered out and replaced with our sound signals, and the emitted beeps can thus be played stealthily. However, even fewer users support this idea since some think playing music while walking actually gets more attention from other people. But only 10% of the participants chose not to support any of these sound candidates. Thus, BumpAlert may provide multiple sound signals for each user to choose based on his preference. Utilizing different sound signals can also enable multiple users to run BumpAlert simultaneously, where the received signals from different users can be differentiated by the corresponding filters. BumpAlert can also use inaudible sounds to detect objects with newer mobile devices that are equipped with better-fidelity microphones/speakers, thus causing no disturbance to users. Next, we present this light modification of BumpAlert based on our evaluation results and user feedbacks.

2.8 BumpAlert+

From the participants' feedback after using BumpAlert, we found most users favoring less user interference, such as running the detection in background, no audible noise, and low false detection over high detection accuracy. For example, they prefer to turn off the object-detection function in a high false positive (e.g., crowded) area rather than getting many false and correct detections. Moreover, while most of the participants in our study liked the benefits of BumpAlert, only 48% of them were happy with the sound signal (of 11kHz). BumpAlert relies on 11kHz beeps to sense environments because it provides the best sensing capability among the mobile devices we tested. Inaudible sound of 22kHz with

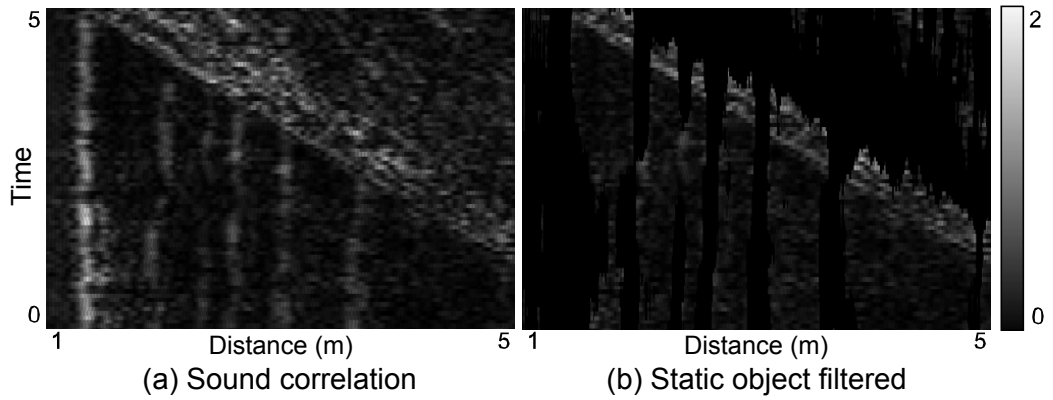


Figure 2.14: Acoustic detection in BumpAlert+. Bright areas indicate the possible existence of detected obstacles.

Galaxy S4/5 is unable to detect objects within a 2–4m range, because the signals captured at 22kHz are significantly weaker than those in the audible range [81]. To preserve the safety of distracted walkers without annoyance, we design and implement an extended system called `BumpAlert+` which provides reasonable detection accuracy with nearly zero user annoyance. `BumpAlert+` is designed as a background system service which uses only an inaudible sound to sense environments. In a crowed area, `BumpAlert+` will not check the image taken by rear camera but pop up a warning message asking users to take care by themselves, and temporarily turns off the detection. The detection range is shrunk to 3m since many participants in our study regarded the detection of objects 3m away as false detections. Currently, `BumpAlert+` can only be executed on Galaxy Note 4 as it provides the highest sensing capability of inaudible signals among the devices we tested. We believe that the design of both `BumpAlert` and `BumpAlert+` can be improved and generalized for devices that will likely emerge in the near future.

The main modification employed in `BumpAlert+` is to use 25ms-long 18kHz–24kHz chirps sampled at 48kHz to sense the environmental reflections. We choose the chirp signal instead of a pure tone since we need to boost the SNR of received signals in this inaudible band. To make this sound inaudible to humans, `BumpAlert+` also applies similar fade-in/out windowing at the beginning and the end of each chirp as shown in [81]. Our experimental results show the signal to noise ratio (SNR) of this particular sound de-

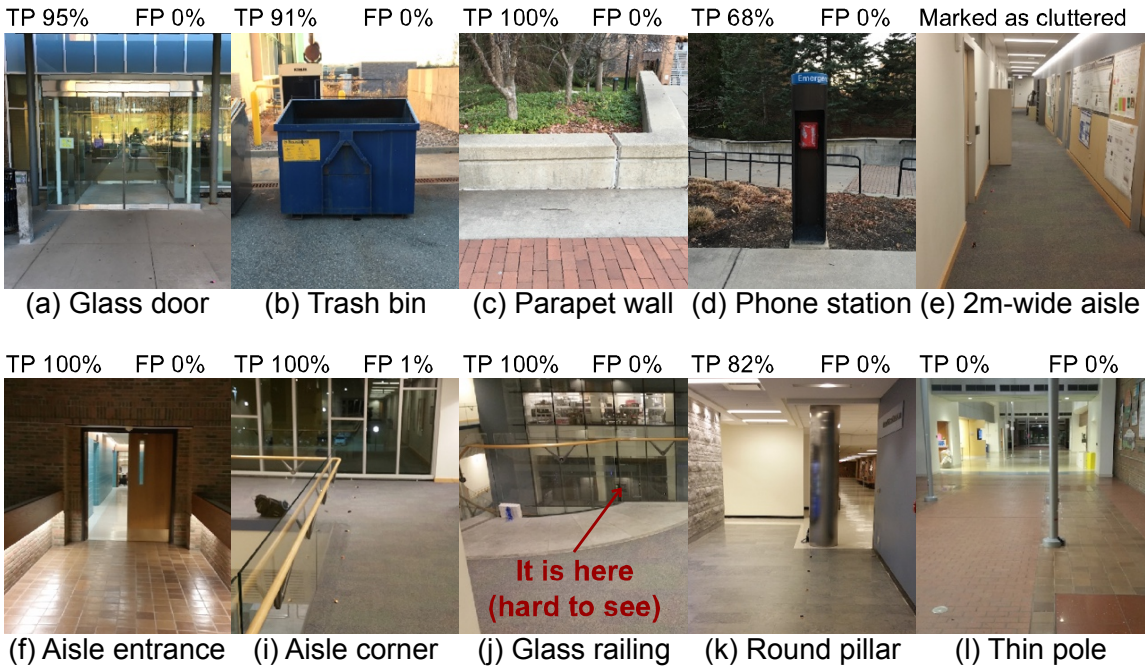


Figure 2.15: Performance of BumpAlert+. Various scenarios have been tested by walking toward the obstacles from a position 10m away.

sign on Galaxy Note 4 provides sufficient signal strength to detect nearby objects. Porting `BumpAlert+` to other devices with compatible hardware settings is part of our future work. In `BumpAlert+`, each chirp sensing period is decreased from 100ms to 50ms since the detection range is set smaller and the audio frequency is higher (so the reflections from far objects decay more quickly). This chirp signal setting also provides less estimation errors and finer granularity due to the property known as *pulse compression* [112].

Based on this new audio setting, the acoustic detector is modified as follows. First, instead of estimating distance by using the highest correlation peak, a one-time calibration is done by sending 10 repetitions of a wide-band pilot signal before using `BumpAlert+`. This calibration process compares the received and the sent pilots, and ends when the microphone/speaker sample offset is tuned to less than 5. After getting the matched filter results as shown in Section 2.3, a time-varying gain is applied to compensate for the decay of the signals reflected from far objects. This is accomplished by multiplying a dynamic gain, i.e., $gain(x) = x^{1.65}$ where x is the audio sample offset. An example of this new detection when

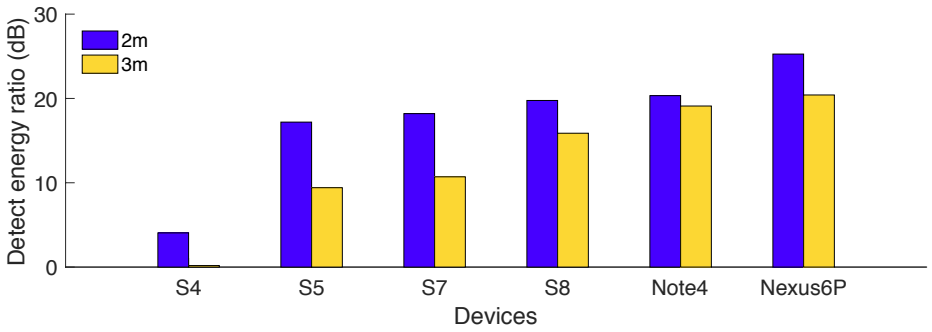


Figure 2.16: Device compatibility. Detect energy ratios are measured as the peak received acoustic energy when the target is present versus the peak energy in an open area without obstacles.

the user is walking toward the corner of an aisle is plotted in Fig. 2.14. This figure can be regarded as a higher-resolution version of Fig. 2.3(c), where the bright areas represent the likelihood of an object detection. As shown in this figure, we reuse the clutter filter to remove objects with speed 0 relative to the user (such as the ceiling or side walls) before applying the motion filter. After removing those objects, we set the threshold to 0.12 in order to alarm users if the median of motion filtered area exceeds this threshold.

Our measurements show that `BumpAlert+` yields comparable results as `BumpAlert` in identifying objects for the scenario shown earlier. We also tested many other objects in both open and crowded areas, and plotted the results in Fig. 2.15. Thin objects like flat poles are invisible to `BumpAlert+` and aisles with width less than 3m are just marked as consistent warning. These results can be further improved by setting a more aggressive threshold. For example, setting the threshold to 0.08 can make the round pillar and the phone station detectable with 98% accuracy with only 4% false positive rate. However, as mentioned earlier, `BumpAlert+` is designed to remove/mitigate users' annoyance, and hence the parameter setting is tuned to ensure a low false positive rate with high priority. This result shows that `BumpAlert+` serves this design purpose, providing reasonable detection accuracy with nearly zero user disturbance. A demo video of `BumpAlert+` can be found from [9].

As mentioned earlier, this *inaudible* optimization is tuned mainly based on Note 4, and

different devices might have varying results of using the same setting. Fig. 2.16 shows the device capability of using `BumpAlert+` to detect a 1.5m-high parapet wall when it is 2 or 3m away from users. The peak detect energy ratio is used to characterize its capability of detecting objects. For example, when the wall is 3m away from users, we first calculate the peak of the reflected signal strength between 2.8m and 3.8m and then divide this value by the peak detection energy in the same range of a reference data collected without any obstacle. This metric represents the required strength of the acoustic reflections to be captured by the device hardware. As shown in Fig. 2.16, Note 4 can receive more than 19dB peak detect energy ratio from the inaudible reflections when the object is 3m away, while S4 only captures less than 5dB even when the object is 2m away. Among the devices we tested, Nexus 6p can provide the best result with `BumpAlert+`. We also notice that the detection capability of Samsung Galaxy S-series devices has improved over time, i.e., S8 $>$ S7 $>$ S5 $>$ S4. Based on our testing results, the current setting of `BumpAlert+` can be applied to S8 and Nexus6p easily. Repeating our previous tests on different devices, like detecting different objects when users are moving, is part of our future work.

2.9 Conclusion

We have explored how to reduce the accident rate of distracted walking by using only phone sensors. A prototype called `BumpAlert` has been designed, implemented and evaluated as a mobile app to warn distracted pedestrians of imminent collision with obstacles. Since `BumpAlert` relies only on built-in sensors of commodity phones, it can be easily deployed on different platforms. `BumpAlert` detects obstacles by fusing several sensor inputs with minimal computation and energy overheads. In the current implementation of `BumpAlert`, the accuracy of detecting objects in front of the user is higher than 95% in both outdoor and indoor environments. This high detection rate of `BumpAlert` is achievable in a wide spectrum of real-life environments, ranging from glass doors to small dustbins, since it does not depend on any *a priori* knowledge of detected objects. Our users study has

shown BumpAlert to be acceptable to the general public and a light-weight version called BumpAlert+ is also proposed based on the users' feedback on BumpAlert. We expect BumpAlert and /or BumpAlert+ will reduce accidents caused by distracted walking.

CHAPTER III

EchoTag

3.1 Introduction

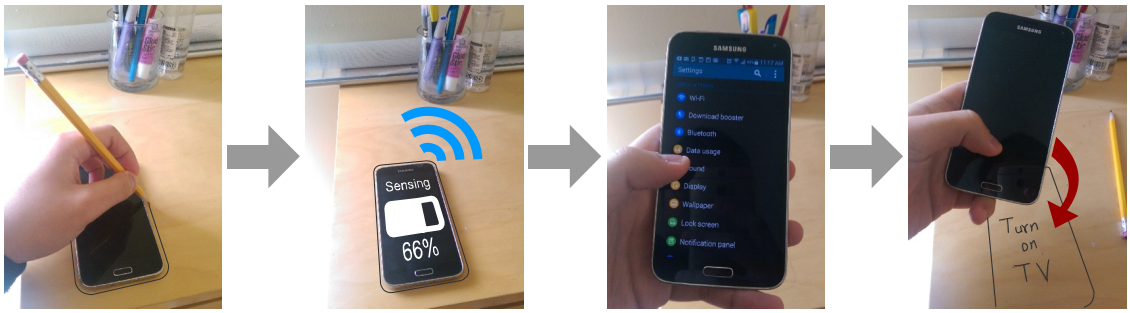
Imagine one day, the silent mode of a phone is automatically activated in order to avoid disturbing a user's sleep when the phone is placed near the bed. Likewise, favorite songs are streamed to speakers whenever the phone is placed near a stereo or a predefined timer/reminder is set if the phone is near a medicine cabinet. This kind of applications is known as context-aware computing or indoor geofencing which provides a natural combination of function and physical location. However, such a function–location combination is still not pervasive because smartphones are not yet able to sense locations accurately enough without assistance of additional sensors or pre-installed infrastructure.

Existing localization systems are unable to provide this type of functionality for two reasons. First, they usually rely on *passively* recorded WiFi, FM, or background acoustic signals, and can only achieve about room- or meter-level accuracy. Nevertheless, the above-mentioned applications need more accurate location sensing, e.g., both streaming music and setting silent mode might take place in the same room or even on the same table as shown in Fig. 3.1. Second, more accurate (i.e., with error of a few cm) location sensing with light recording or acoustic beacons requires a pre-installed infrastructure. The cost of such an infrastructure and the ensuing laborious calibrations make its realization expensive or difficult, especially for personal use.

(a) Auto silent mode (b) Auto music (c) Auto timer

Figure 3.1: Candidate applications of EchoTag. Silent mode is automatically activated when the phone is placed on a drawn box, named *(echo) tag*, near the bed. Favorite songs are streamed to speakers or a predefined timer is automatically set when the phone is placed at other nearby tags.

In this paper, we propose a novel location tagging system, called `EchoTag`, which enables phones to tag and remember indoor locations with finer than 1cm resolution and without requiring any additional sensors or pre-installed infrastructure. The main idea behind `EchoTag` is to *actively* render acoustic signatures by using phone speakers to transmit sound and phone microphones to sense its reflections. This *active* sensing provides finer-grained control of the collected signatures than the commonly-used passive sensing. For example, `EchoTag` emits sound signals with different delays at the right channel to enrich the feature space for sensing nearby locations, and exploits the synchronization between the sender and the receiver as an anchor to remove interferences/reflections from objects outside the target area/locations. Moreover, this active sensing relies only on built-in sensors available in commodity phones, thus facilitating its deployment. Note that `EchoTag` is not designed to replace any localization system since it can only remember the locations where it had been placed before, rather than identifying arbitrary indoor locations. However, this fine-grained location sensing for remembering location tags can enable many important/useful applications that have not yet been feasible due to large location sensing



(a) Draw (echo) tags (b) Sense acoustic signatures (c) Select mapped applications (d) Replay recorded tags

Figure 3.2: Four steps of using EchoTag. The user first draws the contour of target locations/areas with a pencil, then commands the phone to sense the environment. After sensing the environment, a combination of applications and functions to be performed at this location is selected. Finally, the user automatically activates the selected applications/-functions by simply placing his phone back within the contoured area. The contoured areas are thus called *(echo) tags*.

errors or the absence of pre-installed infrastructure.

Fig. 3.2 demonstrates a 4-step process to set up and use `EchoTag`. The first step is to place the phone at the target location and draw the contour of the phone with a pencil. This contour is used as a marker for users to remember the target location, which is called an *(echo) tag*. (Tags can also be drawn on papers pasted on target locations.) Then, `EchoTag` generates and records the surrounding signatures of this target location. The next step is to select the applications/functions being combined and associated with this tagged location. Finally, users can easily activate the combined applications/functions by placing their phones back in the drawn tags. In summary, `EchoTag` embeds an invisible trigger at physical locations that the phone remembers what to do automatically.

We have implemented `EchoTag` as a background service in Android and evaluated the performance by using Galaxy S5 and other mobile devices. Our experimental evaluation shows that commodity phones equipped with `EchoTag` distinguish 11 tags with 98% accuracy even when tags are only 1cm apart from each other and achieves 90% accuracy based on the trace collected a week ago. Since `EchoTag` only utilizes existing sensors in commodity phones, it can be easily implemented and deployed in other mobile platforms. For example, we also implemented and evaluated `EchoTag` on iOS (with traces classi-

fied in Matlab) but omitted the results due to space limitation. Our usability study of 32 participants also shows that more than 90% of participants think the sensing accuracy and prediction delay of EchoTag are useful in real life. Moreover, about 70% of the participants agree that the potential applications in Fig. 3.1 can save time and provide convenience in finding and activating expected functions.

This paper makes the following four contributions:

- The first indoor location tagging achieving 1cm resolution by using commodity phones;
- Demonstration of the ability of active sensing to enrich the feature space and remove interferences;
- Implementation of EchoTag in Android without any additional sensors and/or pre-installed infrastructure; and
- Evaluation of EchoTag, showing more than 90% accuracy even a week after locations were tagged.

The remainder of this paper is organized as follows. Section 3.2 discusses the related work in indoor location sensing. Section 3.3 provides an overview of EchoTag. Sections 3.4–3.6 describe the design of acoustic signature and classifiers. The implementation details are provided in Section 3.7, and the performance of EchoTag and its real-world usability are evaluated in Sections 3.8 and 3.9, respectively. We discuss future directions in Section 3.10 and conclude the paper in Section 3.11.

3.2 Related Work

Indoor localization is a plausible entry to location tagging. The existing localization systems are summarized in Table 3.1. The most popular methods used for indoor localization, such as Radar [38] and Horus [129], sense locations based on WiFi-signal degradation. Their main attractiveness is the reliance on widely-deployed WiFi, hence requiring a minimal deployment effort. However, severe multipath fading of WiFi signals makes WiFi-signature-based localization achieve only room-level accuracy. To overcome the instability

System	Resolution	Infrastructure	Signature
SurroundSense [37]	room-level	No	Fusion
Batphone [116]	room-level	No	Sound
RoomSense [109]	300cm	No	Sound
Radar [38]	400cm	Existing	WiFi
Horus [129]	200cm	Existing	WiFi
Geo [46]	100cm	No	Geomagnetism
FM [44]	30cm	Existing	FM
Luxapose [77]	10cm	Additional	Light
Cricket [103]	10cm	Additional	Sound/WiFi
Guoguo [86]	6–25cm	Additional	Sound
`EchoTag`	**1cm**	**No**	**Sound**

Table 3.1: Existing indoor location sensing systems.

of WiFi signatures and increase the accuracy of indoor localization, researchers have also explored other sources of signatures. For example, the authors of [46] adopted the readings of geo-magnetism which varies with location due to the disturbance of steel structure of buildings. FM radio is also adopted to increase the sensing accuracy of WiFi-based localization [44]. Batphone [116] determines the room locations by sensing the background acoustic noise. Unfortunately, even with these improvements, localization systems relying on passive sensing of the environment can only achieve meter-level resolution. Moreover, passively sensing the environment suffers greatly when the environment changes. For example, as shown in [116], the signature of background acoustic noise changes dramatically when the climate control (HVAC) system was shut off for maintenance, and WiFi RSSI is known to change significantly when the transmit power control in an Access Point is enabled [44].

As shown in Table 3.1, a few systems, such as Luxapose [77], Cricket [103] and Guoguo [86], provide indoor localization with a few cm resolution, thus enabling accurate location tagging. However, these fine-grained localization systems can only be realized with the help from a pre-installed infrastructure. For example, Luxapose requires replacement of ceiling lamps by programmed LED and Guoguo & Cricket require customized WiFi/acoustic beacons around the building. Even if the cost of each additional sensor

might be affordable, the aggregated cost and the laborious calibration required for this deployment are still too high to be attractive/feasible for real-world deployment. Even with the pre-installed infrastructure, the localization error is still around 10cm. The localization error can be reduced further by using antenna or microphone arrays [59, 128], but these advanced sensors are not available in commodity phones. In contrast, EchoTag achieves location sensing with 1cm resolution without any pre-installed infrastructure or additional advanced sensors. Location tagging can also be realized by deploying NFC tags [24], but EchoTag makes this functionality realizable in all commodity phones — such as HTC butterfly, the latest Xiaomi 4, and iOS phones[1] — that are commonly equipped with microphones and speakers, but not NFC chips. Note that EchoTag is *not* designed to replace any localization system since it can only remember the locations where it had been placed before, rather than identifying arbitrary indoor locations like [77, 86, 103]. However, as the results shown in this paper and the feedbacks from the participants of our usability study, this fine-grained location sensing for remembering tags can be used to enable many important/useful applications that have not been feasible before, due to large location sensing errors or lack of installed infrastructure.

EchoTag senses locations based on acoustic signatures. Acoustic signals have been studied widely since they are readily available in commodity phones. For example, SurroundSense [37] and Auditeur [96] classify user behaviors based on background noise. Skinput [63], TapSense [62] and SufaceLink [56] provide new commuter–human interactions based on acoustic reflections from human skin, fingertip, or contacted surface. UbiK [124] provides a input method with acoustic signatures in response to touching different positions on a table. All of these use similar acoustic signatures (e.g., resonances) as in EchoTag, but EchoTag is the first to use it for accurate indoor location tagging. Additional novelties of EchoTag include its *active* generation of acoustic signatures and enrichment of signatures by emitting sound signals with different delays.

[1] Apple locks iPhone 6 NFC to Apple Pay:
http://www.cnet.com/news/apple-locks-down-iphone-6-nfc-to-apple-pay/

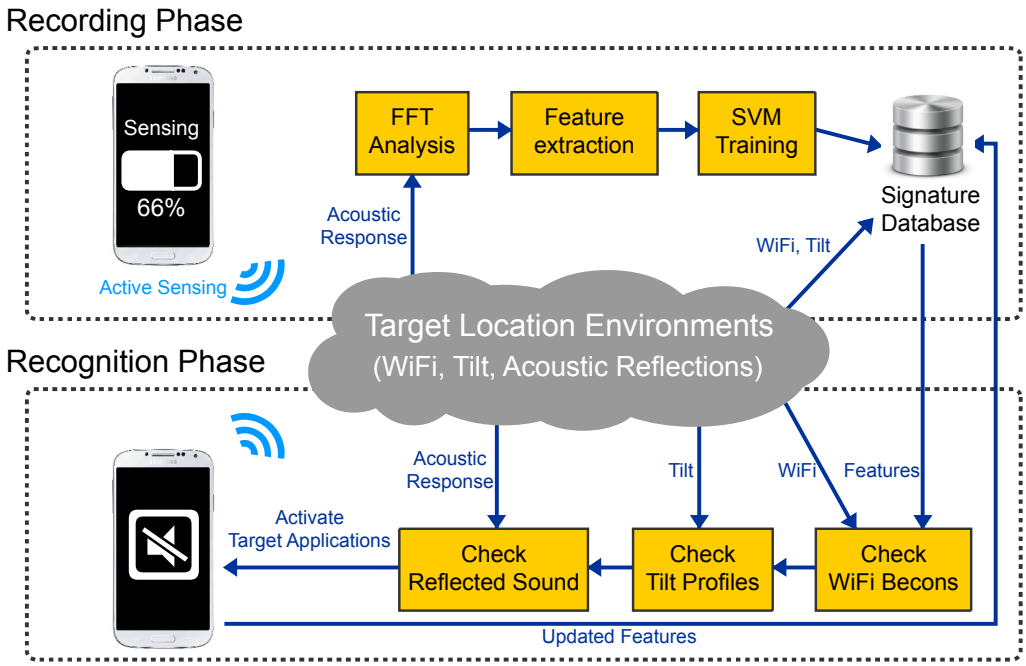


Figure 3.3: System overview. Locations are sensed based on acoustic reflections while the tilt/WiFi readings are used to determine the time to trigger acoustic sensing, thus reducing the energy consumption of the sensing process.

The closest to EchoTag are Touch & Active [97], Symbolic Object Localization [76], and RoomSense [109], all of which also actively generate acoustic signals and record their signatures but for different purposes. Touch & Active [97] uses the same multipath signature to identify how the user touches an object equipped with piezo speakers and microphones. Commodity phones were mentioned as a potential interface for Touch & Active, but no evaluation was provided. The authors of [76] use sound absorption by the touched surface as a feature to identify symbolic locations of a phone — i.e., in a pocket, on a wood surface or a sofa — which is unable to detect nearby locations on the same surface. RoomSense [109] also uses the sound reflections from environments to identify different rooms. However, since only the compressed analytical feature (e.g., Mel Frequency Cepstral Coefficient) is used, its sensing resolution is larger than $9m^2$, thus becoming unable to distinguish nearby tags. In this paper, we implement a novel accurate location tagging system based on actively generated acoustic signatures via built-in phone sensors.

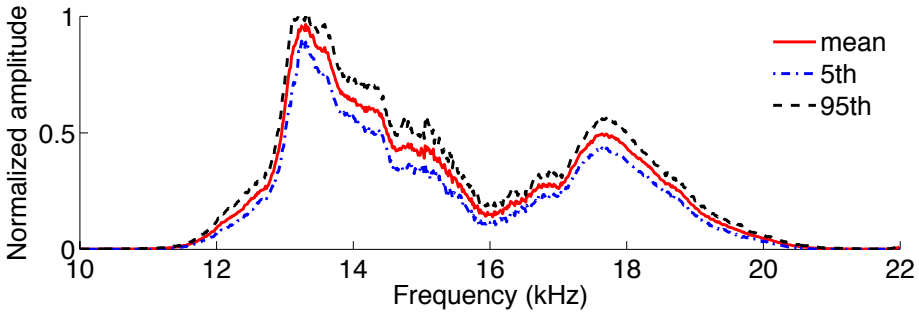


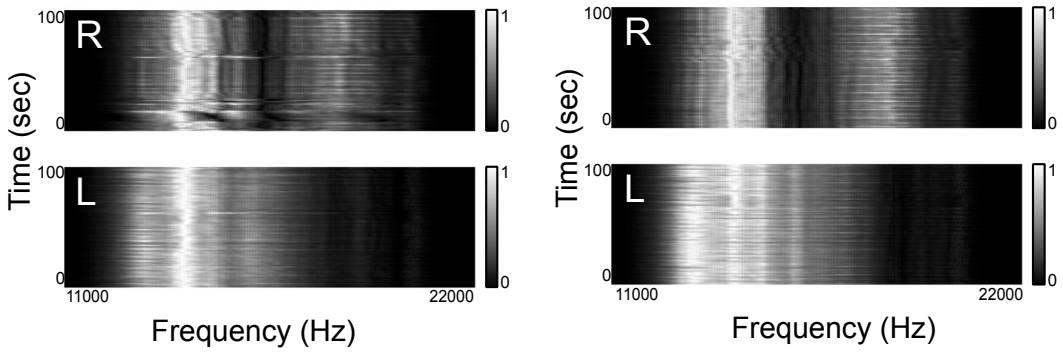
Figure 3.4: An example of acoustic signatures. The received attenuation of a flat frequency sweep is uneven over different frequencies. The result is an average of 100 trials over 1 minute.

3.3 System Overview

Fig. 3.3 gives an overview of `EchoTag` which is composed of recording and recognition phases. In the recording phase, multiple short sequences of sound signals will be emitted from the phone speakers. Each sequence is repeated a few times with different delays between left and right channels to enrich the received signatures as we will discuss in the following sections. The reading of built-in inertial sensors is also recorded for further optimization. After recording the signature, the selected target application/function and the collected signatures are processed and saved in the device's storage. In the recognition phase, the phone will continuously check if the WiFi SSID and the tilt of the phone match the collected signatures. If the tilt and WiFi readings are similar to one of the recorded target locations, then the same acoustic sensing process is executed again to collect signatures. This new collected signature is compared with the previous records in the database using a support vector machine (SVM). If the results match, the target application/function will be automatically activated.

3.4 Acoustic Signature

`EchoTag` differentiates locations based on their acoustic signatures, characterized by uneven attenuations occurring at different frequencies as shown in Fig. 3.4. Note that



(a) Location A (b) Location B

Figure 3.5: Frequency responses at nearby locations. Responses varies with location (i.e., the distribution of light and dark vertical lines) thus being used as a feature for accurate location tagging.

EchoTag does not examine the uneven attenuations in the background noise but those in the sound emitted from the phone itself. For example, as shown in Fig. 3.4, the recorded responses of a frequency sweep from 11kHz to 22kHz are not flat but have several significant degradations at certain frequencies. The characteristics of this signature at different locations can be observed in Fig. 3.5 where the phone is moved 10cm away from its original location. In what follows, we will unearth the causes of this phenomenon and describe how to exploit this feature for EchoTag's accurate location tagging.

3.4.1 Causes of Uneven Attenuation

There are three main causes of this uneven attenuation: (a) hardware imperfection, (b) surface's absorption of signal, and (c) multipath fading caused by reflection. As shown in Fig. 3.6, when sound is emitted from speakers, hardware imperfections make the signal louder at some frequencies and weaker at other frequencies. These imperfections have been identified and used as a signature to track people's smartphones for the purpose of censorship [136]. After the emitted sound reaches the surface touched by the phone, the surface material absorbs the signal at some frequencies. Different materials have different absorption properties, thus differentiating the surface on which the phone is placed [76]. Then, when the sound is reflected by the touched surface and the surrounding objects, the

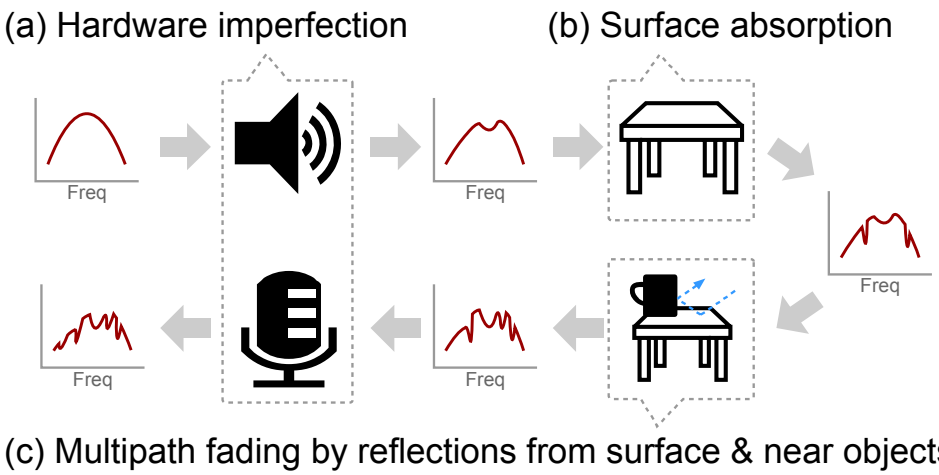


Figure 3.6: Causes of uneven attenuation. During the recording of emitted sound, hardware imperfection of microphones/speakers, absorption of touched surface materials and multipath reflections from nearby objects incur different degradations at different frequencies. Only the degradation caused by multipath reflections is a valid signature for sensing locations even in the same surface.

combination of multiple reflections make received signals constructive at some frequencies while destructive at other frequencies. This phenomenon is akin to multipath (frequency-selective) fading in wireless transmissions. For example, if the reflection of an object arrives at microphones t milliseconds later than the reflection from the touched surface, then the signal component at $10^3/2t$ Hz frequency of both reflections will have opposite phases, thus weakening their combined signal. This multipath property of sound has been shown and utilized as a way to implement a ubiquitous keyboard interface [124]. When reflections reach the phone's microphone, they will also degrade due to imperfect microphone hardware design.

For the purpose of accurate location tagging, `EchoTag` relies on the multipath fading of sound among the properties mentioned above as this is the only valid signature that varies with location even on the same surface. In what follows, we will introduce the challenges of extracting this feature and describe how `EchoTag` meets them.

3.4.2 Sound Selection

The selection of parameters for the emitted sound is critical for EchoTag to extract valid multipath signatures. According to the guideline of Android platforms,[2] 44.1kHz is the most widely supported sample rate for Android phones, so the highest frequency that can be sensed is about 22kHz. Studies have shown that humans can hear signals of frequency up to 20kHz [102]. It is thus desirable to make the emitted sound inaudible (to avoid annoyance) by sensing 20 to 22kHz. But from our preliminary experiments on commodity phones, we found that the signal responses in this high-frequency range are not strong enough to support accurate indoor location tagging due to the imperfect hardware design that causes significant degradation of signal strength in this high-frequency range. Based on the experiments in [81], certain phones' microphones receive signals with 30dB less strength at 22kHz. This phenomenon is even worse if the degradation of speakers is accounted for. Thus, we choose the chirp (i.e., frequency sweep) from 11kHz to 22kHz to sense locations. The frequency response below 11kHz is not used since it contains mostly background noise of human activities [116]. Even though this selection makes the sensing of EchoTag audible to humans, the impact of this selection is minimal because EchoTag triggers acoustic sensing very infrequently, i.e., only when the tilt and the WiFi readings match its database as shown in Fig. 3.3. Moreover, the annoyance caused by sensing with audible sounds is mitigated by reducing the sound volume (e.g., to 5% of the maximum volume) without degrading sensing accuracy. None of the 32 participants in our usability study considered the EchoTag's emitted sound annoying and 7 of them didn't even notice the existence of emitted sound until they were asked to answer related questions in the post-use survey.

We must also consider the length of the emitted sound, which is correlated with the signal-to-noise-ratio (SNR) of received signals. The longer the sound of a frequency sweep, the more energy at each frequency is collected. However, a long duration of emitted sound

[2] http://developer.android.com/reference/android/media/AudioRecord.html

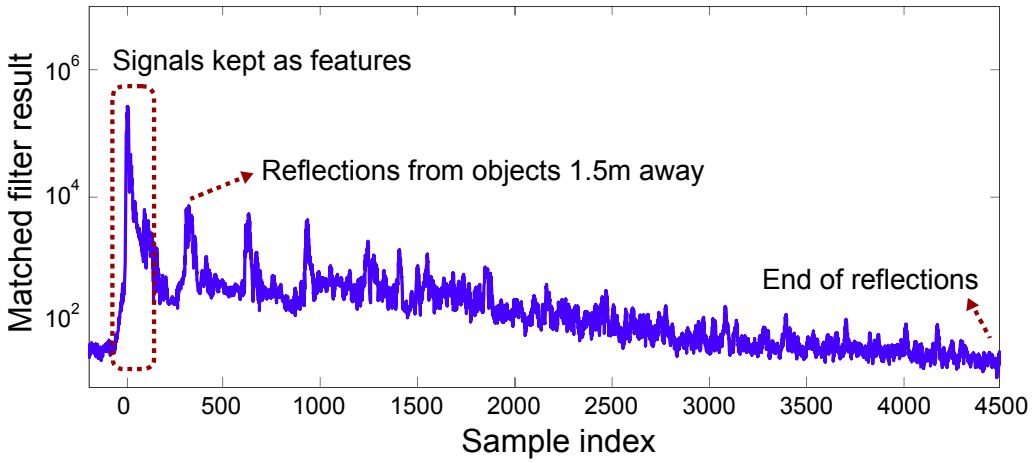


Figure 3.7: Characteristics of reflections. A matched filter is used to identify the reflections of a 100-sample chirp. Only first 200 samples after the largest peak are kept as a feature in `EchoTag`, excluding reflections from objects farther than 86cm away.

introduces a serious problem to the functionality of `EchoTag` because reflections from far-away objects are collected during this long duration of sensing. Fig. 3.7 shows the received signal passed by a matched filter, where the peaks indicate received copies of the emitted sound. The first and largest peak in this figure represents the sound directly traveled from the phone's speakers to its microphones and the subsequent peaks represent the reception of environmental reflections. As the purpose of `EchoTag` is to remember a specific location for future use, it is unnecessary to collect signatures of reflections from far-away objects since those objects are likely to move/change. For example, the object 1.5m away shown in Fig. 3.7 might be the reflection from the body of a friend sitting next to the user, and he might move away when `EchoTag` is triggered to sense locations. One way to mitigate this problem is to truncate the signals generated by the reflections from far-away objects. `EchoTag` uses 100-sample short signals for sensing and collects only the 200 samples of received signals after the largest peak passes through the matched filter. That is, the sensing range of `EchoTag` is roughly confined to 1m since the sound of speed is 338m/s and the chosen sample rate is 44.1kHz. The sensing range is actually shown to be shorter than this calculated value since the signals reflected from nearby objects are inherently stronger than those from far-away objects. Our results also confirm that this setting can meet most

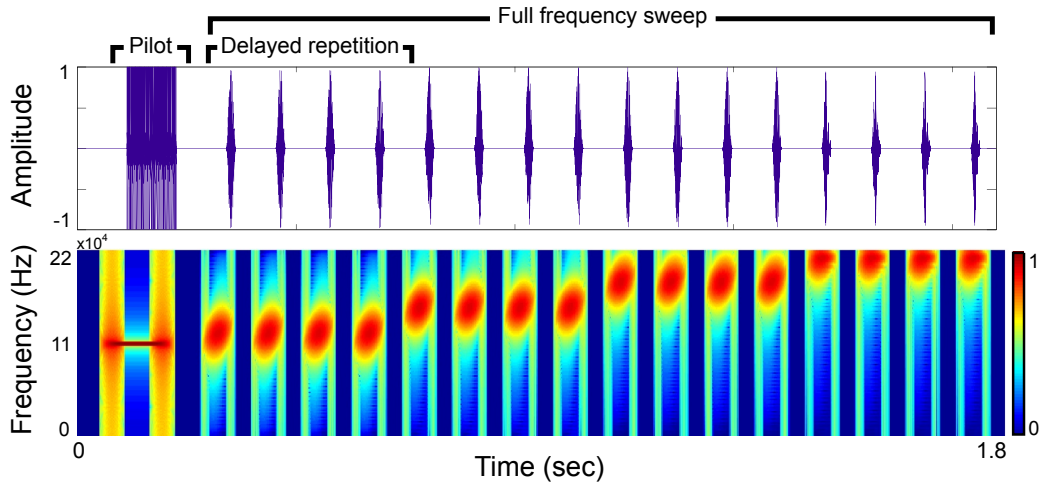
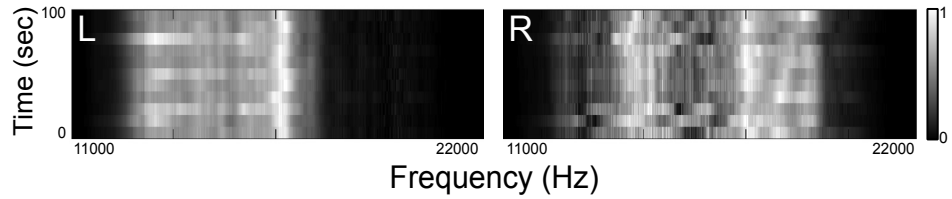


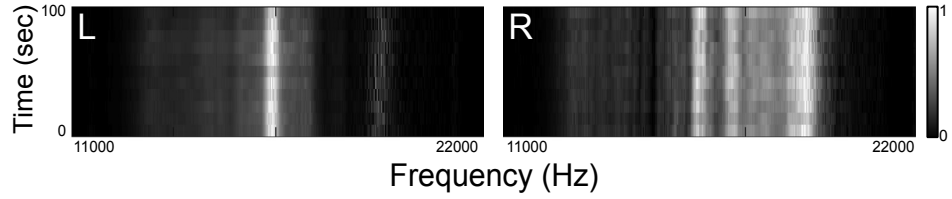
Figure 3.8: Selected sound signals at EchoTag. The leading pilot is used for time synchronization between speakers and microphones. The following chirps (repeated 4 times each) cover the frequency sweep from 11 to 22kHz. (This figure is scaled for visualization.)

real-life scenarios in terms of accurate location tagging. One thing to note is that the entire frequency sweep is divided into four smaller 100-sample segments rather than one 100-sample chirp covering the 11–22kHz range. This selection reduces the sample duration (also the sensing range), but keeps enough energy at each frequency for the purpose of sensing locations.

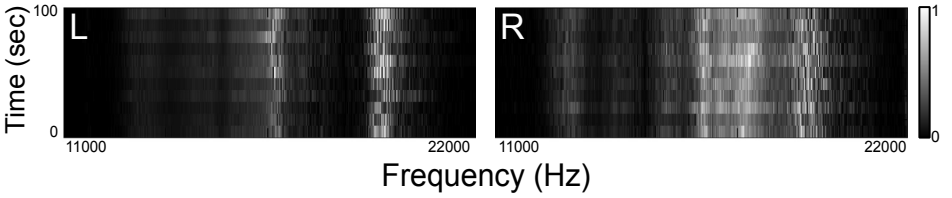
The last parameter in selecting the emitted sound is the time to wait for playing the next chirp after sending a chirp. This parameter is related to the sensing speed of `EchoTag`. The larger this parameter, the longer the time `EchoTag` needs for single location sensing. On the other hand, a short wait time causes detection errors since the received signals might accidentally include the reflections of the previously emitted chirp. For example, if `EchoTag` triggers the next chirp within the 500-th sample shown in Fig. 3.7, the peaks (i.e., reflections) near the 400-th sample will be added as a noise to the received signals associated with the sensing of the next chirp. From our earlier field study to identify the surrounding objects via sound reflections, we found the speakers and microphones on Galaxy S4 and S5 are able to capture the reflections from objects even 5m away. This phenomenon can also be found in Fig. 3.7; there is residual energy even after the 1500-th sample. Thus, the



(a) 100% volume



(b) 5% volume



(c) 1% volume

Figure 3.9: Frequency responses at different volumes. Responses of full volume are saturated by sound directly transmitted from speakers while responses at 1% of the maximum volume are too weak to pick up valid features.

interval between two chirps in `EchoTag` is set to 4500 samples, making its signal sensing time of the entire frequency sweep equal to $4(200+4500)/44100 \cong 0.42$ second.

An example of sensing signals is shown in Fig. 3.8, where a 500-sample pilot is added before the frequency sweep. This pilot is used for synchronization between speakers and microphones because the operating system delays are not consistent in commodity phones. The way `EchoTag` synchronizes a microphone and a speaker is similar to the sample counting process in BeepBeep [100]. In the current version of `EchoTag`, this pilot is set as a 11,025Hz tone, which can be improved further by pulse compression [81, 112], but according to our test results, it doesn't make any noticeable difference. Another 10000 samples follow the pilot before the chirp signals are played. Note that 4 chirps in the same frequency range are played consecutively before changing to the next frequency range. This repetition is used to enrich the acoustic feature space as described in the following sections. Current setting of `EchoTag` makes the total sensing time of `EchoTag` near 2–3 seconds. After test-

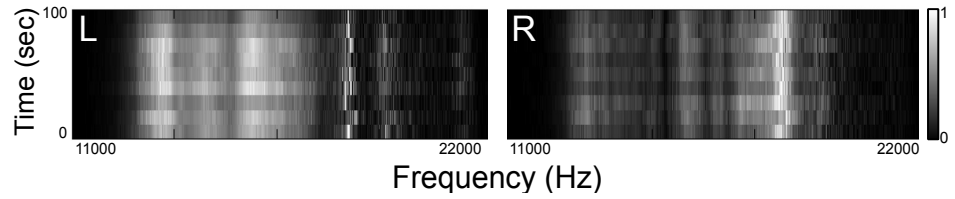
ing EchoTag, most participants of our usability study were satisfied with this latency in sensing locations. Note that in the training phase of EchoTag, each trace is collected with 4 cycles of the above-mentioned frequency sweep to eliminate transient noises, consuming about 10 seconds to collect.

3.4.3 Volume Control

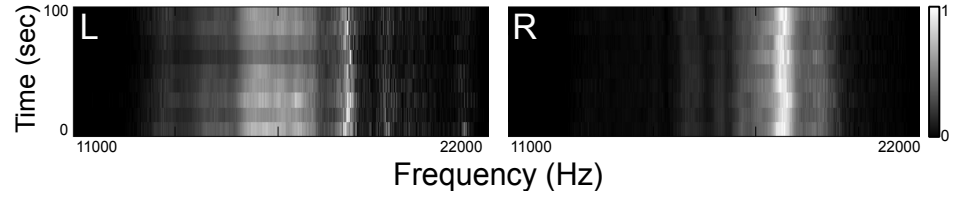
The volume of an emitted sound plays a critical role in extracting valid signatures from multipath fading. As shown in Fig. 3.9, when the volume of emitted sound is full (i.e., 100%), a large portion of the feature space is saturated by the sound emitted directly from the phones' speakers. Moreover, emitting sound in full volume makes the sensing process more annoying to the users since EchoTag uses audible frequency ranges. On the other hand, if only 1% of full volume is used to emit sound, the reflections are too weak to be picked up by phones' microphones. Based on our preliminary experiments, setting the phone volume at 5% is found optimal for Galaxy S5. Even though this setting varies from one phone type to another, calibration is needed only once to find the optimal setting.

3.4.4 Acoustic Signature Enrichment

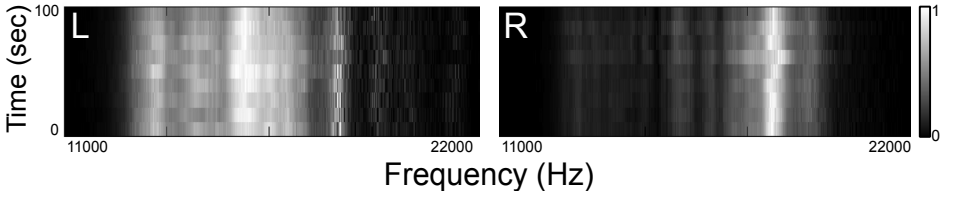
The goal of EchoTag is to enable accurate location sensing with fine resolution, but according to our experimental results, one shot of the frequency sweep between 11 and 22kHz can distinguish 1cm apart objects with only 75% accuracy. One way to enrich the feature space is repeating the emitted sound which can be used to eliminate the interference caused by transient background noise. Instead of only repeating the emitted sound, EchoTag also adds delay of emitted sound in the right channel at each repetition. This intentional delay at the right channel is designed for two purposes. First, when there are stereo speakers in the phone, such as HTC M8 and Sony Z3, this intentional delay yields an effect similar to beamforming in wireless transmission, which helps us focus on the response in one specific direction at each repetition. We validated this feature to enrich the



(a) 0 sample delay



(b) 1 sample delay



(c) 2 samples delay

Figure 3.10: Frequency responses with delay at the right channel. When the emitted sound is intentionally delayed at the right channel, different portions of features are strengthened, which helps enrich the feature space for sensing locations.

collected signatures by HTC M8's two front-faced speakers. A similar concept was also adopted in acoustic imaging [69], but `EchoTag` doesn't need calibration among speakers because the purpose of this delay is used to enrich the feature space rather than pointing to a pre-defined target direction. Second, the intentional delay also helps strengthen features at certain frequencies even when there is only one speaker in the phone. The effects of this delay at the right channel are shown in Fig. 3.10, where different portions of features are *highlighted* with different delays. Based on the results in Section 3.8, 4 repetitions with 1 sample delay at the right channel improve `EchoTag`'s sensing accuracy from 75% to 98% in sensing 11 tags, each of which is 1cm apart from its neighboring tags. This way to enrich the acoustic signature space is a unique feature of `EchoTag`, as it *actively* emits sound to sense the environment, rather than passively collecting existing features.

3.4.5 Modeling of Sensing Resolution

Suppose two (drawn) tags are at distance d from each other, the sensing signal wavelength is λ, and the angle from one of the nearby objects toward these two tags is θ. The difference of the reflection's travel distance from this object to the two tagged locations is $\delta = 2d * cos\ \theta$. Since the sensing signal wavelength is λ, a change, $\delta > \lambda/2$, in any reflection's travel distance will cause the summation of all acoustic reflections to vary from constructive to destructive combing (or vice versa), thus resulting in a significant change in the acoustic signature. So, if θ of all nearby objects is not close to 0 (which is also rare in the real world), tags separated by more than $\lambda/4$ are to be distinguished by their acoustic signatures. Based on this model, the current setting of `EchoTag` is capable of identifying tags with a 1cm resolution. Our evaluation also validates this property as shown in the following sections. However, this fine resolution also implies that users should place their phones close enough to the trained location for collecting valid signatures; this is also the reason why `EchoTag` requires "drawn" tags to remind users where to place their phones. In our usability study, most users didn't have any difficulty in placing phones back at the trained locations with this resolution to activate the tagged functionality of `EchoTag`. The limitations and future direction of this resolution setting will be discussed further in Section 3.10.

3.5 Classifier

Several classifiers, such as k-nearest neighbors (KNN) and support vector machine (SVM), have been tried for location sensing based on acoustic signatures. Our experimental results show that one-against-all SVM [122] performs best in classifying locations. For example, in the experiment of sensing eleven 1cm apart tags based on the training data collected 30min earlier, 98% accuracy can be achieved by SVM while only 65% test data can be correctly classified via KNN with the Euclidean distance and $k = 5$. We believe

this inaccuracy is caused by the small training data size and nonlinear nature of acoustics signatures. For example, a 5mm position change might cause more significant feature changes (in the frequency domain measured by the Euclidean distance) than a 10mm position change since the acoustic signatures capture the superposition of all reflections.

In the one-against-all SVM, n classifiers are trained if there are n tags to sense. A location is classified as the tag k if the k-th classifier outputs the highest prediction probability for that tag. In our test with 2-fold cross validation, the linear kernel achieves the optimal performance. As the results shown in Section 3.8, the difference of prediction probability between the classifiers trained at the target location and the other locations is greater than 0.5 in most cases, which is adequate for `EchoTag` to distinguish locations using acoustic signatures.

3.6 Performance Optimization

Even though activating microphones and speakers is shown to be more economical than image or WiFi sensing [39], the cost of continuously collecting acoustic signals is still non-negligible and unnecessary. Our measurements with Monsoon Power Monitor [23] on Galaxy S5 show that acoustic sensing consumes 800mW. Moreover, due to the constraints in existing phone microphones, the signals we used are still audible, especially for the pieces of frequency sweep close to 10kHz. The strength of acoustic signal is reduced greatly by lowering the volume, but its continuous use still annoys the users and consumes energy. We have therefore performed further optimizations by avoiding unnecessary acoustic sensing to reduce the power consumption and user annoyance. For example, in terms of `EchoTag`'s functionality, it is useless to sense the environment via acoustic signals when the user keeps the phone in his pocket while walking. This situation can be easily detected by reading inertial sensors. As shown in Fig. 3.3, `EchoTag` first checks the status of the surrounding WiFi to ensure that the phone is located in the same target room. Then, the inertial sensor data, such as the accelerometer readings, are used to check if the phone is

placed with the same angle as recorded in the database. If both the WiFi status and the inertial readings match the recorded data, one shot of acoustic sensing will be activated to collect the surrounding acoustic signature. The next round of acoustic sensing can be executed only when EchoTag finds the phones moved and the recorded WiFi beacons and inertial readings match. Note that WiFi sensing in EchoTag incurs minimal overhead, since it only needs connected WiFi SSID, which can be directly retrieved from the data already scanned by Android itself via WifiManager. In the current implementation of EchoTag, tilt monitoring only consumes additional 73mW in the background and the power usage of WiFi is negligible since EchoTag uses only the (already) scanned results. We will later evaluate the false trigger rate of this design.

3.7 Implementation

We implemented EchoTag as an Android background service. Since EchoTag relies only on sensors commonly available in smartphones, it can be readily implemented on other platforms like iOS or Windows. Acquisitions of active acoustic signatures, tilts, and WiFi signals are implemented with Android API while the classifier relying on LIBSVM [43] is written in C layered by Java Native Interface (JNI). The function to trigger applications, such as setting silent mode or playing music, is implemented via Android Intent class. A prototype of EchoTag is also implemented in iOS with classifiers trained in Matlab. In our current implementation, one round of acoustic sensing (including SVM processing) takes about 4 seconds. Most participants in our usability study are satisfied with the current setting.

3.8 Performance Evaluation

We have conducted a series of experiments to evaluate the performance of EchoTag using Galaxy S5 for two representative scenarios shown in Fig. 3.11. Certain experiments

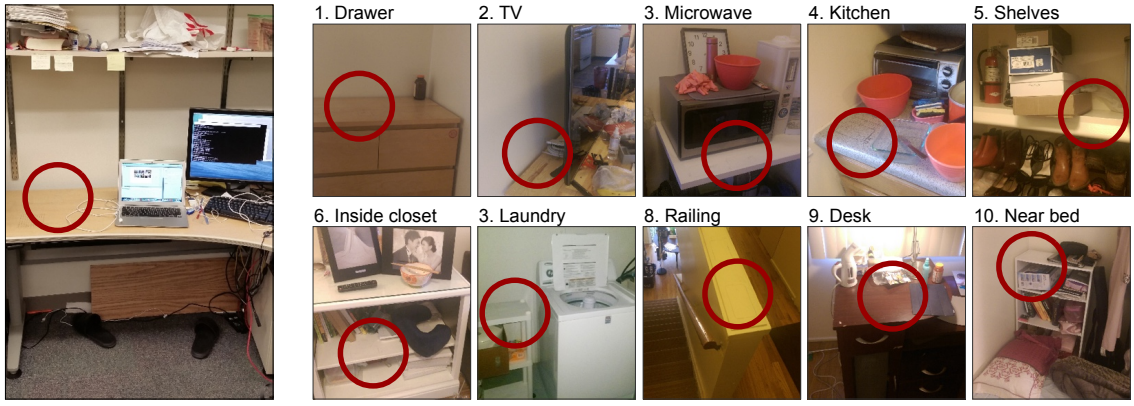


Figure 3.11: Experiments scenarios. Red circles represent the target location to draw (echo) tags.

are also repeated on Galaxy S4/Note3, and iPhone4/5s, but the results are omitted due to space limitation. The first test environment is a lab/office and the second is a two-floor home. The red circles in Fig. 3.11 represent the test locations to draw tags. Both scenarios represent real-world settings since people usually work or live in either of these two environments. In the lab/office environment, traces were collected while lab members were chatting and passing through the test locations. During the experiment, an active user kept on working (changing the laptop position and having lunch) on the same table. There are two residents living in the home environment, and one of them is unaware of the purpose of our experiments. The residents behave normally, cooking in the kitchen, watching TV, and cleaning rooms. Thus, our evaluation results that include the interference due to human activities should be representative of real-life usage of `EchoTag`.

In the lab/office environment, three tag systems shown in Fig. 3.12 are used to evaluate the *sensing resolution* which is defined as the minimum necessary separation between two tags. The first system is composed of three tags 10cm apart: A, B, and C, as shown in Fig. 3.12(a). This setting is used to evaluate the basic resolution to support applications of `EchoTag` (e.g., automatically setting phones to silent mode). The second and third systems include 11 tags which are 1cm (30°) apart from each other as shown in Fig. 3.12(b) (Fig. 3.12(c)); this is used to evaluate the maximum sensing resolution of `EchoTag`. In the

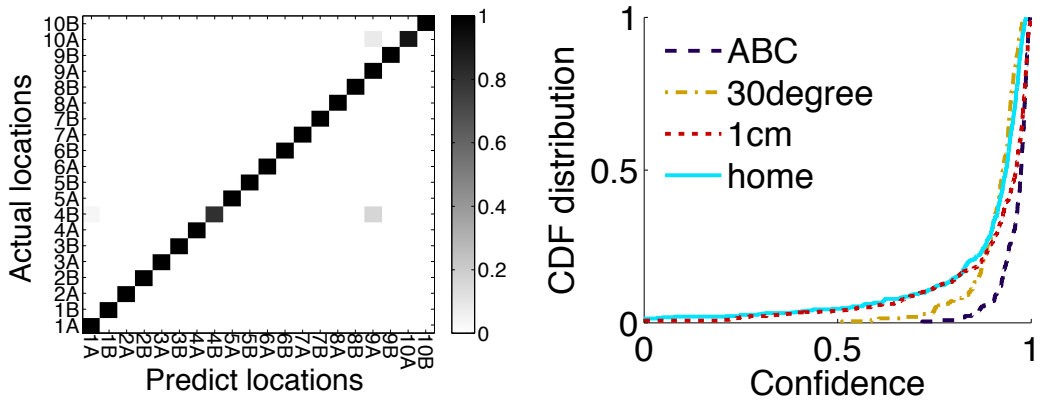
(a) ABC tags (b) 1cm tags (c) 30° tags

Figure 3.12: Tag systems. The first tag system consists of disjoint (echo) tags while the second and third tag systems are composed of overlapped tags 1cm or 30° apart.

home environment, 10 locations are selected as shown in Fig. 3.11(b). At each location, we marked two tags, A and B, similar to the setting in Fig. 3.12(a).

In both scenarios, traces are collected at different sampling frequencies and time spans, generating three datasets: 1) 30min, 2) 1day, and 3) 1week. In the 30min dataset, traces are collected every 5 minutes for 30 minutes, which is used to evaluate the baseline performance of `EchoTag` without considering the environment changes over a long term. The 1day dataset is composed of traces collected every 30 minutes during 10am – 10pm in a day. The purpose of this dataset is to evaluate the performance changes of `EchoTag` in a single day, which is important for certain applications, such as the silent-mode tag near the bed where the users place their phones every night. The last 1week dataset is collected over a week, which is used to prove the consistency of active acoustic signatures over a one-week period. In the lab/office environment, the 1week dataset is sampled during 9:00am – 9:00pm every day while the 1week dataset in the home environment is tested at 11:00pm.

We believe these experiments can validate the effectiveness of `EchoTag` in real world. Larger-scale experiments including more users and spanning longer periods are part of our future plan after deploying `EchoTag`.



(a) Confusion matrix in home

(b) Confidence distributions

Figure 3.13: Result of 30min dataset. *Confidence* is defined as the prediction probability at the target location minus the largest prediction probability at the other locations.

3.8.1 Accuracy and Resolution

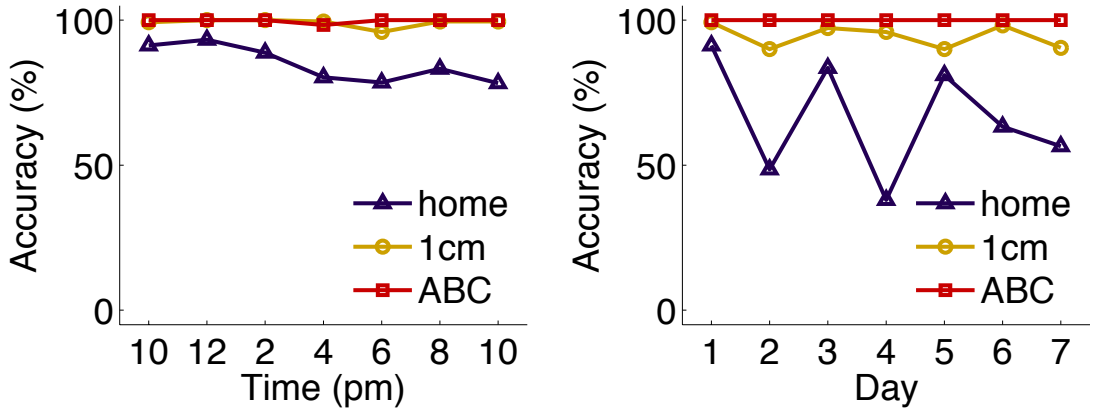
Sensing resolution in `EchoTag` is defined as the minimum necessary distance/degree between tags, which is an important metric since it is related to the number of tags that can exist in the same environment. On the other hand, the sensing accuracy at a location is defined as the percentage of correct predictions at that location, whereas the overall accuracy is defined as the average of accuracy at all locations. In the lab/office environment of 30min dataset, the average sensing accuracy under all settings is higher than 98%. Orientation changes can be detected by `EchoTag` since the microphones/speakers are not placed in the middle of the phone, and hence the relative position changes when the phone is rotated. `EchoTag` can also distinguish 20 tags in a home environment with 95% accuracy. The resulting confusion matrix is shown in Fig. 3.13(a). This evaluation based on the 30min dataset validates that `EchoTag` can achieve a sensing resolution of 1cm and at least 30°. Without using any infrastructure, this sensing resolution is the finest among existing methods to differentiate locations. Our measurements show that WiFi RSSI or background acoustic noise can only distinguish the 20 tags at home with 30% accuracy.

3.8.2 Uniqueness and Confidence of Acoustic Signature

To measure the uniqueness of acoustic signature, we define a metric called *confidence* as the prediction probability of the classifier trained at the target location minus the largest prediction probability of classifiers at other locations. A high confidence means high feature uniqueness among locations since SVM gets less confused among locations. A prediction is wrong whenever the confidence is less than 0 because SVM will choose another location with the highest prediction probability as the answer. Fig. 3.13(b) shows the confidence distribution of 30min dataset in all environments. ABC tags get the highest confidence since the tags are separated by more than 10cm and only 3 tags are considered. However, even 20 tags are set at home or tags in office are separated by only 1cm (overlapped), acoustic signatures are still distinct enough to differentiate 90% of cases with confidence greater than 0.5. This example demonstrates that the uniqueness of active acoustic signature is good enough to support the function of `EchoTag`.

3.8.3 False Positives

The above-mentioned accuracy represents the true positive rate to differentiate locations. To prevent `EchoTag` from falsely classifying locations without tags as tagged ones, two more traces are recorded on the same surface but 10cm away from each tag. These traces are used to build an additional *No Tag* SVM classifier which determines if the sensed traces belong to one of tagged locations or not. We set 0.5 as the threshold that any sensed location is classified as *No Tag* when the prediction probability of this classifier is greater than 0.5. In the 30min dataset of home environment, the probability to classify random locations without tags as tagged locations (i.e., false positive rate) is only 3%, and this setting only causes a 1% false negative rate. We conducted another test by deploying the ABC tags on three users' office desks for three days. The users carry their phones as usual but are asked not to place their phones inside the drawn tags. In this test, merely 22 acoustic sensings are triggered per day and only 5% of them are falsely classified as being placed at tags



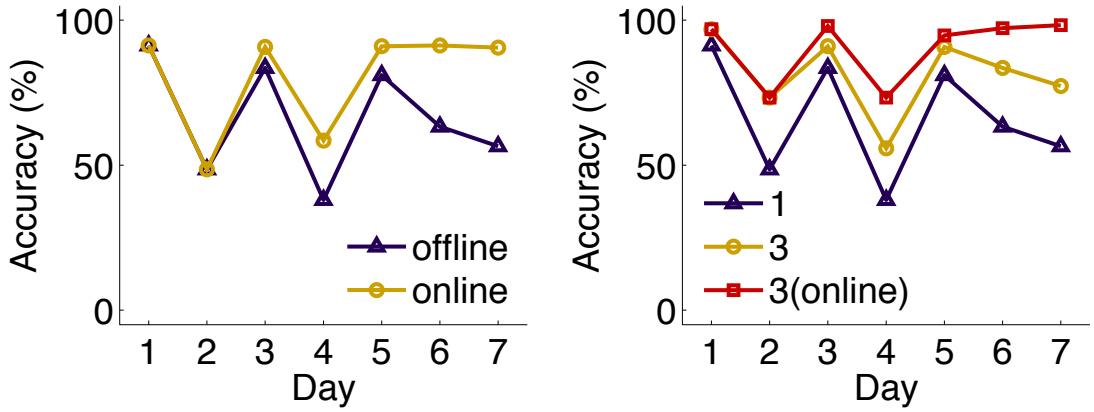
(a) 1day dataset (b) 1week dataset

Figure 3.14: Accuracy variation over time/day. Prediction is based on 6 traces collected during the first hour/day.

with online-updated classifiers. This rate can be reduced further by fusing other signatures which is part of our future work. We also implemented a manual mode (without any false trigger) in which users can press the home button to manually activate acoustic sensing. This manual mode is a reasonable design choice since it is similar to the way Apple Siri or Google Search is triggered.

3.8.4 Temporal Variation

The purpose of this evaluation is to test how active acoustic signatures change over time. This is an important metric since `EchoTag` is designed to be able to remember the location at least for a certain period of time. To evaluate this, the average accuracy among tags of 1day and 1week datasets based on the first 6 traces collected in the first hour or day is shown in Fig. 3.14. As shown in Fig. 3.14(a), the decay of active acoustic signatures in one day is not significant. In the lab/office environment, the accuracy drops only by 5% in predicting the traces collected 12 hours later. Even one week later, the average accuracy of `EchoTag` in the lab/office environment is still higher than 90%. However, as shown in Fig. 3.14, the average accuracy of traces collected in the home environment drops by 15% after 12 hours and the traces collected a week later achieve only 56% accuracy.

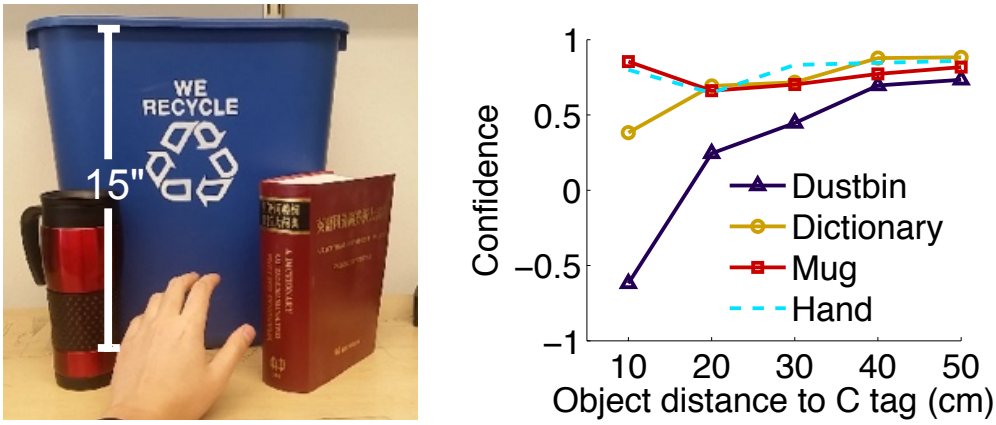


(a) Online SVM (b) Multiple candidates

Figure 3.15: Performance of online SVM and providing multiple candidates in the 1week home dataset. Online SVM classifiers are updated using the traces collected in previous days while the traces collected on the same day are excluded.

This phenomenon is caused by a series of environment changes at certain locations. For example, the accuracy drops mainly at 4 locations: (1) drawer, (4) kitchen, (9) desk, and (10) near the bed, which suffer environment changes due to human activities like cooking in the kitchen or taking objects out of drawers. When the above-mentioned objects are excluded from dataset, `EchoTag` can sense the remaining 12 tags at 6 locations with 82% accuracy even a week later. This result suggests where to put tags is critical to `EchoTag`'s performance. When we consider the tags in the kitchen as an example, if the tags are not placed at the original location near the cooker and stove but on one of the shelves, the acoustic signatures decay as slowly as at other locations. Providing guidelines for where to put tags is part of our future work.

Sensing accuracy over a long term can be improved further in two ways. The first is to use online SVM training. We simulated online SVM by retraining the offline SVM classifiers with the traces collected before the test day (i.e., excluding the same day test data). The cost of retraining classifiers can be further optimized by online SVM [117]. As the results shown in Fig. 3.15(a), with online training, the average accuracy for the home environment in one week can be increased to 91.5% during the last three days. This online training is possible in real world because we assume users will provide feedback, such as



(a) Addition of objects (b) Prediction confidence

Figure 3.16: Test of environmental changes. EchoTag gets less confident when the size of an added object is larger and its position is closer to the test locations.

selecting the right application/functions when a wrong prediction is made or suspending wrongly-triggered applications. By monitoring this user reaction after each prediction, online training data can be collected during the normal use of EchoTag. Moreover, in our experiments, only 8.5% of error predictions need this user interaction.

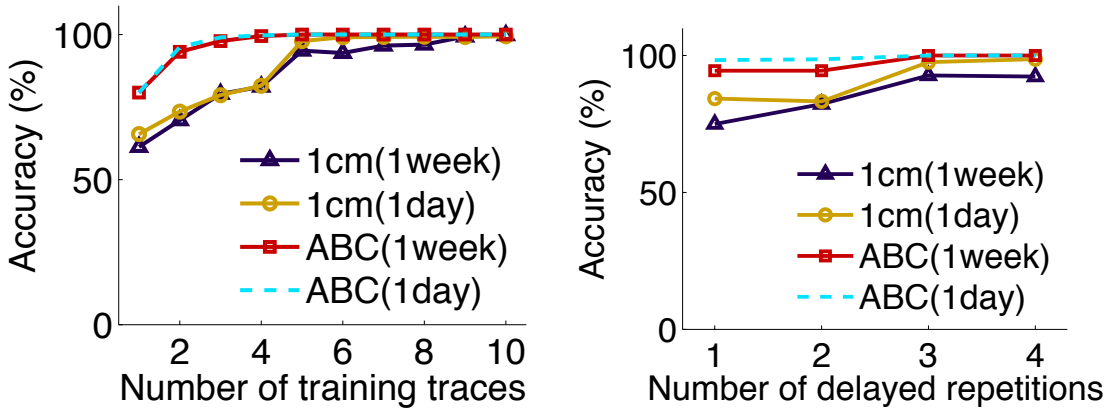
Another way to improve the sensing accuracy over a long term is to provide more than one predicted candidate for users. The candidates in EchoTag are provided based on the prediction probability for each classifier. As shown in Fig. 3.15(b), when the first three predicted candidates are provided, the accuracy during the last day based only on the first day trace is increased to 77%. Moreover, providing 3 candidates with online SVM training boosts the accuracy of EchoTag to 98% during the last day. Evaluating the overhead of users' interactions with online training feedback and multiple candidates is part of our future work.

3.8.5 Environmental Disturbances

Similar to the signature decay due to significant environmental changes in the kitchen, we investigate the performance of EchoTag when objects near a tagged location are different from those during the training. Fig. 3.16(a) shows 4 selected objects: 1) dustbin, 2)

dictionary, 3) mug, and 4) human hands. We add these objects sequentially near the ABC tags and vary their distance to the C tag. The corresponding prediction confidence is used to measure the change of EchoTag's performance. As shown in Fig. 3.16(b), human hands and small objects like a mug cause little disturbance to EchoTag even when those objects are only 10cm away from the test locations. Medium-size objects like a thick dictionary degrade EchoTag's confidence to 0.38 when it is close to the test locations, but most predictions still remain correct. Placing large objects like a 15" high dustbin around the test locations change the acoustic signatures significantly since it generates lots of strong acoustic reflections. Most predictions are wrong (i.e., confidence < 0) when the dustbin is placed 10cm away from the test locations. It is also the reason why the accuracy in the kitchen degrades after the position of a large cooker is changed. However, this large environment change is less common in real life. For example, users may change their mugs or hands frequently but less likely to move large objects on their office desk.

When a large environmental change occurs, EchoTag needs to retrain its classifier to account for this change. One interesting finding from our 1week trace is that the prediction accuracy in the home environment increased back to 90% after three day classifier online updates. This demonstrates that with enough training data, EchoTag is able to keep only *invariant* features. In future we plan to derive a guideline for setting up tags and providing online feedback when EchoTag finds more training necessary for certain locations. Another interesting finding is that the (dis) appearance of human causes only limited effect on EchoTag because human body is prone to absorb sound signals rather than reflect them. During experiments, a human body (the tester) continuously changed his relative position to the test locations, but no significant performance degradation was observed in spite of the large human body.



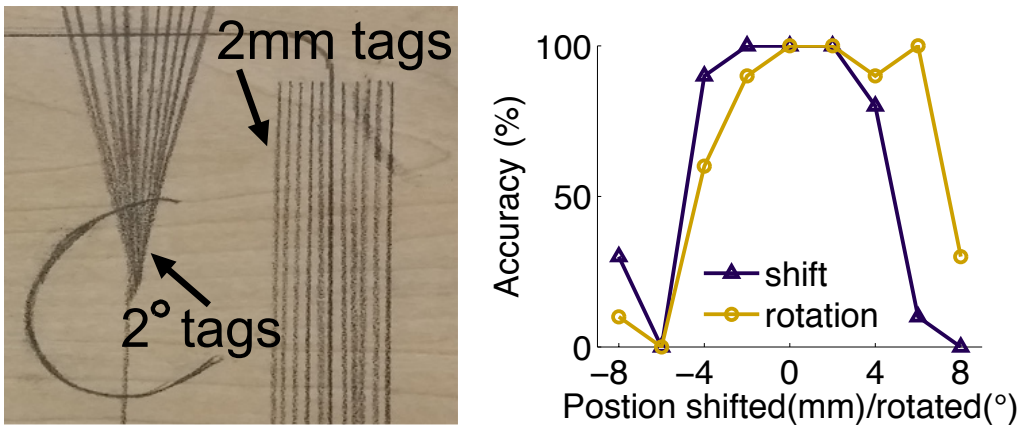
(a) Training traces (b) Delayed repetitions

Figure 3.17: Impact of acoustic feature space. Accuracy is higher than 95% when 5 traces with 4 delayed repetitions are collected.

3.8.6 Acoustic Feature Space

Here we discuss the effect of the feature space selected for sensing locations. We first examine accuracy with different training data sizes. The training data size is relevant to the usability of `EchoTag` since it represents the time required for users to set tags and the number of features necessary to remember a single location. As shown in Fig. 3.17(a), in the lab/office scenario, 2–4 traces are able to distinguish rough locations and only 5 traces can achieve 95% accuracy with 1cm resolution. Considering the tradeoff between training overhead and sensing accuracy, the current version of `EchoTag` sets the size of training data at a single location to 4. In our usability study, these 4 traces at each tag can be collected in 58 seconds on average. We also received a comment from one survey participant that the training process and delay of `EchoTag` are acceptable since the entire procedure is similar to that of setting up Apple TouchID. Based on our informal experiments with 5 participants on iPhone5s, setting up Apple TouchID required 11–13 training traces with different finger placements, taking about 1 minute. See Section 3.9 for details of the users' other reactions on this training set size.

Next, we study the benefit of our delayed repetitions for sensing locations. The sensing accuracy based on 5 traces with different numbers of delayed repetitions is plotted in



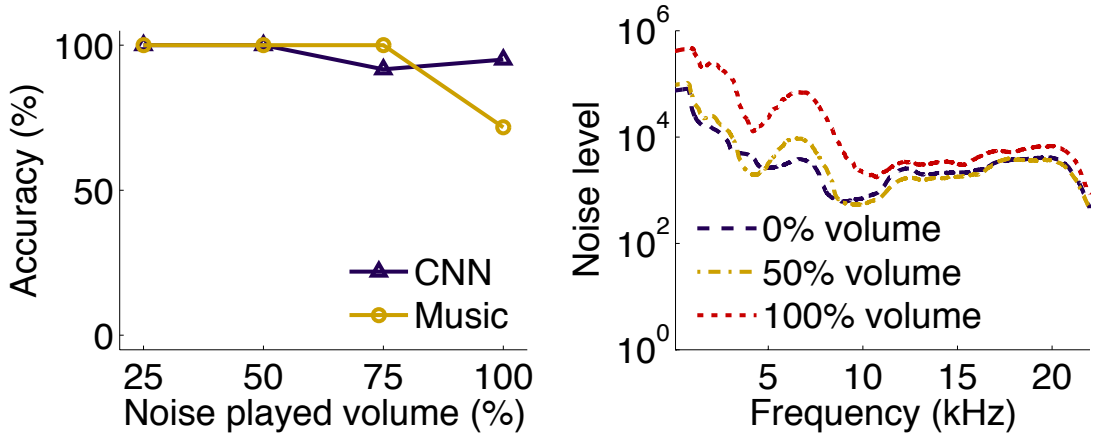
(a) Testbed (b) Accuracy of C tag

Figure 3.18: Tolerance test. Additional tags separated by 2(mm/°) are placed inside the C tag. Test data at C are collected with errors ranging from -8 to 8(mm/°) for knowing the tolerance of `EchoTag`. Dataset of ABC and 1cm tags are combined, so the accuracy shown is the prediction among 14 locations.

Fig. 3.17(b). As shown in this figure, without help of delayed repetitions to enrich acoustic signatures, the accuracy for the 1cm dataset over a week is only 75% while it can be boosted to 91% when 4 delayed repetitions are used. This way of enriching the feature space is only available in active acoustic sensing since we have the full control of emitted/-collected signals. We also find similar effectiveness of this delayed repetitions on phones with stereo speakers like HTC M8, but these results are omitted due to space limitation. The current version of `EchoTag` uses 4 delayed repetitions.

3.8.7 Tolerance Range

Since the phone might not be placed exactly at the same locations as it had been trained, it is important to know the maximum tolerance range of `EchoTag` and if users can handle the tolerance well. To evaluate the maximum tolerance of `EchoTag`, additional fine-grained tags are drawn inside the C tag as shown in Fig. 3.18(a). These fine-grained tags are separated by 2(mm/°). Test data of these inside tags are collected with additional errors (i.e., between $\pm$8mm/°), and a prediction is made by the first 4 traces of 30min dataset in the lab/office environment. In the training phase, ABC and 1cm datasets are combined so the



(a) Accuracy at different noise

(b) Noise profile of CNN news

Figure 3.19: Impact of background noise. Predefined noises (i.e., music and CNN news) are played by Macbook Air with different volumes. EchoTag is able to provide effective prediction even when the noise is played at 75% of the full volume.

prediction is made for sensing 14 locations. The accuracy of the C tag when test data is collected with additional errors is plotted in Fig. 3.18(b) where the maximum tolerance range of EchoTag is about $\pm$4mm and $\pm4^\circ$. The reason why accuracy with different degree errors is not centered at 0° might be the measurement errors in collecting the training data (i.e., the training data is also collected with a few degree errors). This result also matches our resolution test, where EchoTag can provide 1cm resolution since features of tags separated by 8mm vary significantly. With the help of drawn tags, this tolerance range is good enough to allow users to put their phones back at the tags for triggering EchoTag. Our usability study of 32 participants validates this hypothesis since most of them think it is easy to place phones on the tags. We will discuss how to enhance the user experience with this limitation of tolerance range in Section 3.10.

3.8.8 Noise Robustness

The last issue of EchoTag to address is its ability to sense locations in a noisy environment. Basically, the results discussed thus far were obtained in real-life scenarios where traces were collected in the presence of noises from TV, people chatting, and air condition fans. To further test the robustness against noises, we collected traces in a lab/office envi-

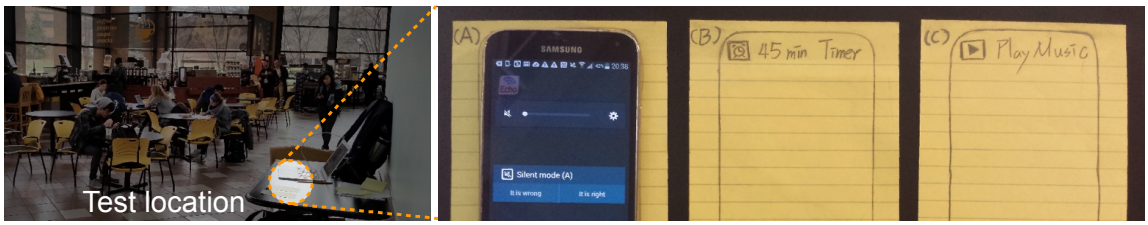


Figure 3.20: Usability study environments. The test location is selected near the a cafe at a student center. Tags are drawn at memo pads since the table is black. Passers by and students studying in this area are randomly selected to test EchoTag.

ronment when a laptop was placed 30cm away from tags on the same surface. This laptop is set to either play a song ("I'm yours – Jason Marz") or a clip of CNN news with different volumes. The traces of ABC and the 1cm dataset are combined to make prediction for 14 locations. The results of this experiment are shown in Fig. 3.19. As shown in Fig. 3.19(a), EchoTag can tolerate the music noise with 75% volume and can perform normally even with noise from CNN news with 100% volume. In our measurements, the intensity of noise from the music with 75% volume and CNN news with 100% volume is about $11dB$ higher than the office background noise. Even though EchoTag is unable to work accurately (i.e., only 71% among 14 locations) when the music was played with 100% volume, this is not a typical operation scenario of EchoTag since it incurs $17dB$ higher noise.

EchoTag is robust to most real-world noises mainly because it relies on signatures actively generated by itself, rather than passively collected from background. That is, as the noise profile of CNN news shown in Fig. 3.19(b), most noise from human speaking occurs in frequencies less than 10kHz while EchoTag uses higher frequencies to sense locations. This phenomenon is also consistent with the results shown in [116]. The current setting of EchoTag is already resilient to the noise in our usual environment. Our usability study done at a noisy location near a cafe also validates the robustness against noise. Incorporating other sources of signatures that are free from acoustic noise is part of our future work.

3.9 Usability Study

In this section, we explore how real users of EchoTag perceive its performance. For example, we would like to answer a question like "can users easily place phones on drawn tags?" To evaluate the usability of EchoTag, 32 (9 female and 23 male) participants were recruited at our university students' activity center. The test location was the table near a cafe as shown in Fig. 3.20. Participants are randomly selected from those passing by this area. All participants didn't know EchoTag and its developers, but all have experience in using smart phones (11 Androids, 19 iPhones, and 2 others). Most participants are university students of age 20–29 while 6 of them are not. We first introduced the functionality of EchoTag to users and its three representative applications as shown in Section 3.1. Then, three tags (i.e., turning on silent mode, setting a 45min timer, and playing music) were deployed at the table. Since the table surface is black and is the university's property, we drew the tags on sheets of 5×6 paper which were then attached on the table. Our survey results also validate that drawing tags on attached papers will not affect the accuracy of EchoTag. After users were informed of the purpose of EchoTag, we asked them to put phones on the tags and EchoTag is triggered automatically to make its prediction on attached applications. The participants were asked to first put phones inside the tag for measuring prediction accuracy, and then slightly move the phones away from the tags until a wrong prediction is made for testing the tolerance range of EchoTag. After trying the functionality of EchoTag with the tags we trained, the participants were asked to train their own tags. This process helps users "sense" the amount of effort to set up and use EchoTag, indicating the average user training time. The participants then filled out the survey forms. Each study lasted 15–20 min.

The main results of this usability study are summarized in Table 3.2. 31 of 32 participants think the sensing accuracy of EchoTag is adequate for its intended applications. During this study, three users experienced a bug in our app. In these cases, we restarted the whole survey process and asked the participants to try EchoTag again. The average

Questions	Disagree	No opinion	Agree
Sensing accuracy is useful	1	0	31
Sensing noise is acceptable	0	3	29
Sensing delay is acceptable	1	6	25
Placing phones inside (echo)tags is easy	0	3	29
`EchoTag` can help me remember turning on silent mode when going to sleep	2	5	25
`EchoTag` can help me remember setting the timer for taking washed clothes	5	3	24
`EchoTag` can save my time in activating apps under specific scenarios	1	0	31

Table 3.2: Usability survey results of 32 participants.

accuracy including these three buggy traces were 87%, while the results excluding these three cases were 92%. One thing to note is that even those three users experienced the low sensing accuracy during the first trial, still think the overall detection accuracy of `EchoTag` is acceptable and useful. Moreover, 25 of the participants think the prediction delay is acceptable for the purpose of `EchoTag`. In another question to learn their expectation of `EchoTag`, only 2 of users hope to have the sensing delay less than 1 second and 12 users hope the sensing delay can be shortened to about 2 seconds.

Based on participants' experience in placing phones on tags, 29 of them think it is easy to do. Five participants put their phones too far away from the drawn tags during the first trial, but they were able to place their phones on the tags after they were told of it. Only one participant made a comment that he expected a larger tolerance range. Actually, there is an inherent tradeoff between the tolerance range and the training overhead. For example, a very long training sequence that moves phones at all possible locations near the drawn tags can increase the tolerance range, but with a significant overhead in the training phase. In the current version of `EchoTag`, 4 repetitions are used in the training phase to collect signatures at a single location. Between two consecutive repetitions, the phones need to be taken away from, and then placed back on tags for sensing valid signatures. On average, users need 58 seconds to finish the whole training phase for a tag. 8 participants expected less than 2 training repetitions at each location, while 17 of participants think the

4 repetitions setting is reasonable since it is only one-time training. Considering this trade-off between accuracy, tolerance, and training overhead, the existing setting of EchoTag satisfies most scenarios and users.

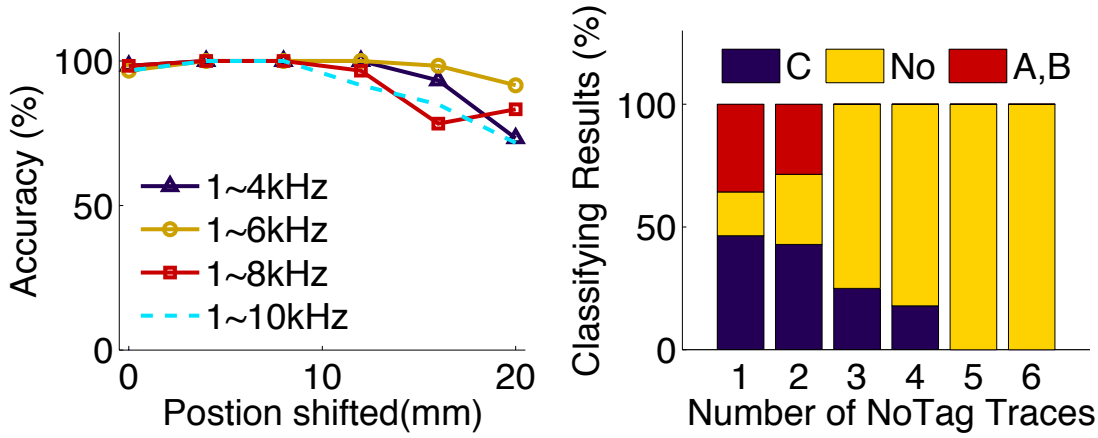
According to the survey related to potential scenarios based on EchoTag, 25 participants agree the silent mode tag can help them remember setting phones to stay quiet during sleep, and 24 of them agree the auto-timer scenario can help them take clothes out of a washing machine. These results indicate that the users see benefits from EchoTag in remembering things to do in specific situations since the functionality of EchoTag provides natural connections between the locations and things-to-do. Moreover, 31 participants also think the automation of triggered applications can save time in finding and launching desired applications.

3.10 Discussion and Limitations

The purpose of EchoTag is to provide a novel way of accurately tagging locations. With the realization of this functionality, we believe many advanced applications can be realized. Most participants in our usability study agree with the scenarios we considered, and they also provide other possible uses of EchoTag.

3.10.1 Potential Applications

Similar to those applications mentioned so far, participants also suggest use of an auto-set timer during cooking or auto-set silent mode in an office environment. These applications can be classified into two categories based on the resulting benefits: 1) helping users remember things, and 2) saving time for users in finding and launching desired applications. For example, applications like auto-set silent mode do not save a significant amount of time since pressing a phone's power button can also set up silent mode manually. However, this function turns out to receive most support from the participants because it is easy to forget setting silent mode before going to bed everyday. Instead of doing it manually,



(a) Tolerance in lower frequency

(b) NoTag SVM classifiers

Figure 3.21: Extension of tolerance range. The tolerance range can be extended by sensing tags with lower-frequency signals. Building 'NoTag' classifiers can also prevent EchoTag from incorrect classification of misplacements.

a natural combination of a location (i.e., a tag near bed) and target applications/functions (i.e., silent mode) helps users remember them more easily. Other suggested applications like automatically turning off lights or activating a special app for monitoring the sleep behavior are also in this category. On the other hand, the applications like automatically playing favorite songs (movies) on speakers (TV) help users find the expected application-s/functions with less time due to the hint of tagged locations. For example, we received a feedback saying that EchoTag can be used to set up Google Map and activate the car mode when a tag is placed inside the car (e.g., an iPhone stand). Actually, this task can be easily remembered whenever the user gets into his car, but it is time-consuming to find and enable these applications. In this scenario, the natural combination of locations and appli-cations can help users *filter* out unnecessary information and save their time in triggering the desired applications.

3.10.2 Limitation of Tolerance Range

As shown in Section 3.8, the merit of EchoTag's fine resolution comes with the limi-tation that the tolerance range of current setting is about 0.4cm. Even though most partic-ipants in our user study were able to place phones at the tagged locations accurately after

they were instructed on how to use EchoTag, this fine resolution and its limited tolerance range may not be needed for certain applications and cause unintended prediction errors when the phone is not placed correctly. To meet such applications needs, one can take the following two approaches.

First, we can enlarge the tolerance range of EchoTag (while decreasing its sensing resolution). Based on the mathematical model in Section 3.4.5, we can lower the sensing frequency to extend the tolerance range. For example, setting the highest sensing frequency to 6kHz can increase the tolerance range from 0.4cm to about 1.4cm. This increase of tolerance range is plotted in Fig. 3.21(a), showing that lower-frequency signals are better in identifying tags with large placement errors. However, lowering sensing frequency will increase the audibility of sensing signal and decrease the feature space. For example, sensing with 1$\sim$4kHz lowers overall accuracy because the collected signature is not enough to be distinguished from others. Studying the tradeoff between tolerance range and other concerns is part of our future work.

Second, instead of trying hard to classify misplacements as trained tags, EchoTag can simply opt to report "There is no tag" once the location is out of the tag's tolerance range. We choose to use the same 'NoTag SVM' classifier as introduced in Section 3.8.3, which is built from traces in the same surface but at least 0.4cm away from the tags. That is, EchoTag identified locations as "There is no tag" if the trace gets the prediction probability of NoTag classifier which is greater than 0.3 and also larger than the prediction probability of other classifiers. With this method, if the user places his phone outside of the operation range of EchoTag, the message of "There is no tag" (rather than identifying a wrong tag and then triggering a wrong function) will tell the user he needs to place the phone more precisely to activate the tagged function. The result of the same experiment of identifying C tag with traces collected 0.4cm away from the trained location is shown in Fig. 3.21(b). Without the help of 'NoTag' classifier, 50% of those misplaced traces around the C tag were classified as wrong tags, i.e., as A and B. However, with enough NoTag training traces, EchoTag can

identify the misplacements with a high probability, thus not triggering functions attached to wrong tags. Note that NoTag SVM with 6 training traces in this experiment only causes 1% false negatives when the trace is collected at the right location (within the tolerance range), which also matches the result reported in Section 3.8.3.

3.11 Conclusion

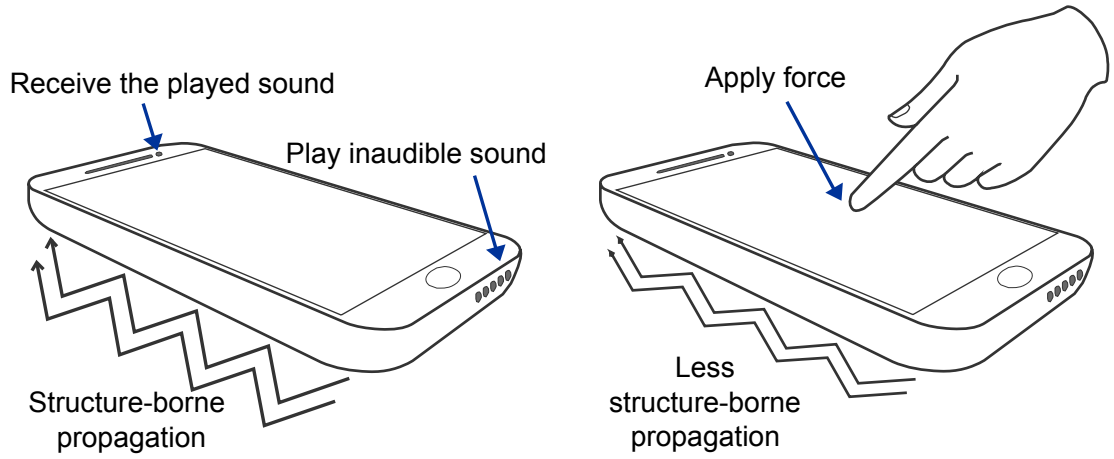
We have proposed and implemented `EchoTag`, a novel indoor location tagging based only on built-in sensors available in commodity phones. `EchoTag` is designed to be able to remember indoor locations with 1cm resolution, enabling the realization of many new applications. The main idea of `EchoTag` is to *actively* sense the environments via acoustic signatures. With the help of active sensing, a fine-grained control of collected signatures can be achieved for either enriching the feature space or removing environmental interferences. Our evaluation in different environments validates the capability of `EchoTag` to meet the users' need. In future, we would like collect larger datasets after deploying `EchoTag`.

CHAPTER IV

ForcePhone

4.1 Introduction

As smartphones become an essential part of our daily activities, human–phone interactions have become a norm. To enhance the input capability on the severely limited space of a phone's touch screen, researchers and practitioners have been seeking various ways to expand the input dimensions. For example, augmenting a force-sensitive, deformable, or squeezable input is shown to enrich the input vocabulary significantly, especially for the one-hand operations [55, 90, 106]. However, most of those extended input interfaces have not yet been fully developed nor deployed in commodity phones for two reasons. First, they usually require additional hardware, such as capacitive or contact piezoelectric sensors [56, 98, 99], which are usually unavailable in commodity phones, and the requirement of additional cost and space make them less attractive to phone users and manufacturers [32]. Second, systems based only on built-in sensors usually impose unnatural/inconvenient usage restrictions because it is challenging to recognize those interactions without additional sensors. For example, users are required to touch the microphone reception hole or block the camera flash light source for sensing a touch interaction [68, 87], both of which limit the usability of this additional sensing. Other systems require continuous vibration of the phone with a vibration motor, causing significant annoyance to users [67]. To the best of our knowledge, the only few commodity phones supporting a force-sensitive



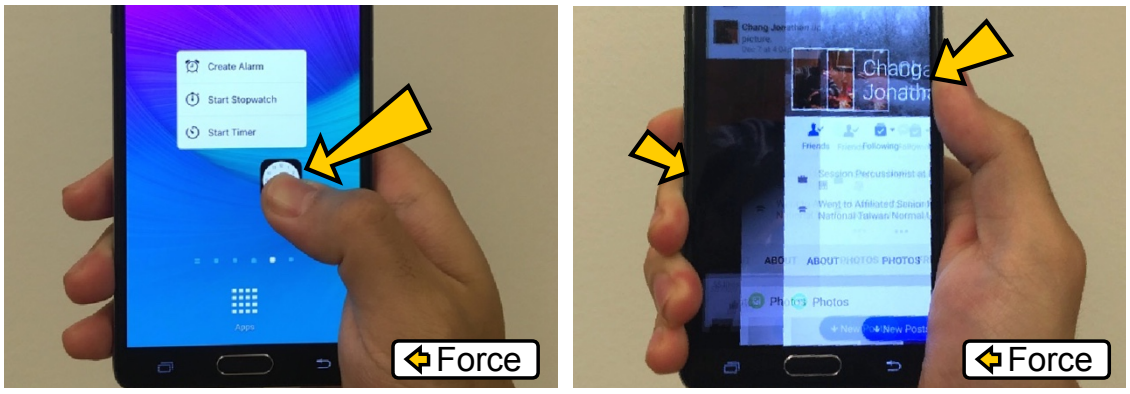
(a) No force (b) Force applied

Figure 4.1: Structure-borne propagation and the applied force. When no force is applied to the phone, the frame and internal components of the phone can vibrate freely, and hence the played inaudible sound can easily propagate through the phone's body.

touch screen are the iPhone 6s and iPhone7, enabled by their proprietary sensors [5]. Unlike these systems, we propose a new, inexpensive solution, called `ForcePhone`, which provides a force-sensitive input interface to commodity phones without any addition/modification of hardware. Moreover, `ForcePhone` provides this force-sensing capability to the touch screen as well as the phone's body, called a *squeezable interface*.

`ForcePhone` estimates the user-applied force by utilizing the structure-borne sound propagation, i.e., the sound transmitted through subtle vibrations of the device body. In most designs, such as headphones or pipe-work, this type of propagation is usually considered as a mechanical noise, but `ForcePhone` uses it in a novel way to estimate the force applied to commodity phones. As shown in Fig. 4.1, when the phone is left free to vibrate (e.g., the user is not touching/squeezing the phone), the sound sent from the phone's speakers can easily travel through its body to its microphone. However, when force is applied to the phone, it restricts the phone body's vibration with the sound, thus degrading the sound traveling through this structure-borne pathway. `ForcePhone` estimates the amount of force applied to the phone by monitoring the degree of this degradation.

As mentioned earlier, it is challenging to provide extended input interfaces with only



(a) Hard-pressed option (b) Squeezable back button

Figure 4.2: Demo apps of ForcePhone. Users can reach an option page when a button is pressed hard and can also surf the previous webpage by squeezing the phone.

limited built-in phone sensors. For example, in `ForcePhone`, the recorded audio includes not only the sound signal transmitted through the phone's body but also that through air, and other reflections. To achieve reliable force estimation, `ForcePhone` exploits the feature of *active* acoustic sensing to get a proper reference point for fetching the signal traveling through the phone's body. `ForcePhone` also utilizes the information of other sensing materials to enhance sensing accuracy and reduce the false detection rate. For example, the location and the start time of a touch are inferred from the touch screen, and the phone's movement is inferred from accelerometer readings, which are then used to *filter* out unexpected audio signal changes caused by environment and human noises. By integrating readings of these sensors, `ForcePhone` can achieve high force-sensing accuracy while limiting the false positive rate.

To date, `ForcePhone` has been implemented on Android and iOS phones to provide a force-sensitive and squeezable interface. We have realized several apps based on `ForcePhone`. Fig. 4.2 presents two examples of those apps which have been shown useful: (1) the hard-pressed option, which is similar to the 3D Touch on iPhone 6s where varying options are given to users when applying different amounts of force to buttons, and (2) the squeezable back of a phone, which allows users to surf the previous webpage by just squeezing the phone. Note `ForcePhone` is not limited to these two use-cases, and can extend

human–mobile interactions in many ways. For example, it has been shown useful to zoom in/out maps by squeezing phones or type upper-case letters by applying force to keyboards [55, 90, 106]. Other potential apps of ForcePhone are discussed in Section 4.7.

In our experiments, the estimated force is shown greater than 0.9 correlation to the real applied force with 54g errors in stationary environments. The participants in our controlled experiments were able to use ForcePhone for applying force at 2 different levels with a higher than 97% accuracy and squeezing the phone body correctly with a greater than 90% accuracy. After trying our demo apps of ForcePhone most participants think our current design comparable to state-of-the-art proprietary sensors in accomplishing simple tasks, such as hard-pressing of a button. A demo video of ForcePhone can be found from [14].

This paper makes the following four main contributions:

- Design of force-sensitive and squeezable interfaces via structure-borne sound propagation;
- Implementation of ForcePhone on both iOS and Android phones;
- Demonstration and evaluation of two popular apps; and
- Achieving a higher than 90% accuracy of using the demo apps in most scenarios.

The rest of this paper is organized as follows. Section 4.2 discusses the related work on expressive input interfaces. Section 4.3 describes the principle of structure-borne propagation, and Section 4.4 details the design of ForcePhone based on this principle. The implementation and evaluation of ForcePhone are presented in Sections 4.5 and 4.6, respectively. Limitations and potential use-cases are discussed in Section 4.7, and the paper concludes with Section 4.8.

4.2 Related Work

Expanding the dimension of touch inputs has been a major research topic for many years owing to the limited-size touch screens of mobile phones. Users are shown to be able to easily and effectively control force-sensitive, deformable, or squeezable input interfaces

System	Interaction	Methods	Hardware	Working Areas
Apple 3D Touch [5]	Force	Force sensors	Additional	Touch screen
Camera and Flash [87]	Force	Camera flash	Existing	Camera lens
Touch&Active [97, 98]	Force	Sound	Additional	Sensors attached objects
PseudoButton [68]	Force	Sound	Existing	Microphone reception hole
Expressive Touch [99]	Tapping material	Sound	Existing	Touch screen
ForceTap [64]	Instantaneous force	Accelerometer	Existing	Phone screen+body
VibPress [67]	Force+Squeeze	Vibration motor	Existing	Vibrated phone body
Acoustruments [79]	Force+Squeeze	Sound	Additional	Hardware attached objects
`ForcePhone`	**Force+Squeeze**	**Sound**	**Existing**	**Phone screen+body**

Table 4.1: Existing touch interfaces to enrich input dimensions.

as they are akin to many natural interfaces, such as applying force to a water knob [55, 90, 106]. A force-sensitive interface can be implemented by adding capacity sensors [106], adding contact piezoelectric sensors [98], checking the flash light source blocked by a human hand [87], monitoring the reduction of sound volume by covering the microphone reception hole [68], and estimating the damped motor vibration with accelerometers [67]. These systems either require additional sensors or impose stringent usage restrictions, such as blocking the flash light source by hand or continuously vibrating the phone.

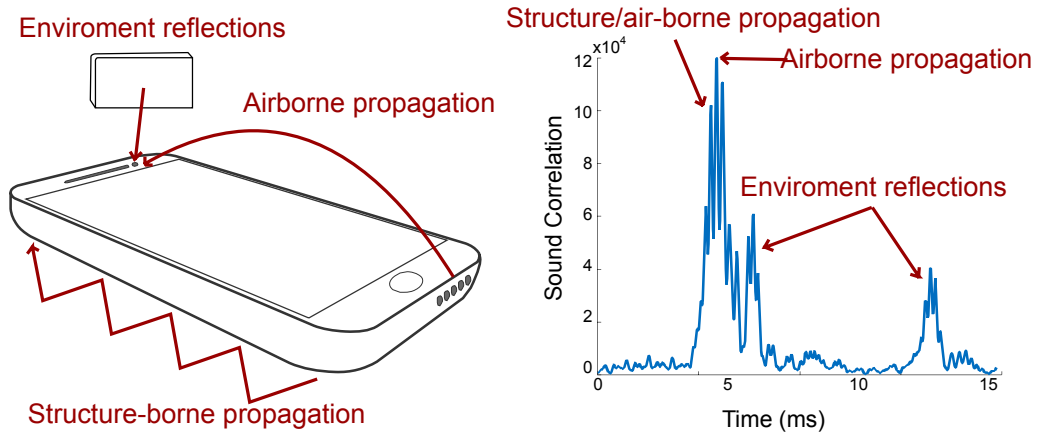
There have also been systems that extend the user's input by recognizing the materials of tapping or the instantaneous tapping force. For example, TapSense [62] classifies the touch sound to recognize if the user touches the screen by a finger tip or a fist. ForceTap [64] utilizes the accelerometer to learn how hard the user taps or shakes the device. It is also possible to extend the touch surface by other accessible objects. SurfaceLink [56] allows users to perform gestures on a table where the phone is placed, while SkinInput [63] uses the human body as an extension of input interface. Here we focus on how to provide a force-sensitive and squeezable input interface by utilizing the structure-borne sound propagation. Table 4.1 shows a comparative summary of `ForcePhone` and other related work. Note that instantaneous tapping force is different from the force sensing introduced in this paper; the former only represents the instantaneous force sensed at the time of the user contacting the touch screen, while the latter continuously monitors the force applied to the phone after the

user touches the phone. Our system is parallel to these instantaneous-tapping-force systems and can be integrated with others to enrich the user experience further.

Sound has been a widely-used sensing method as it only requires microphones and speakers on commodity phones. For example, knowing the surrounding sounds and reflections can provide accurate indoor localization [88, 118] or tracking human living/sleeping behaviors [93, 96]. `ForcePhone` is also sound-based but designed for providing force-sensitive and squeezable interactions. The closest to `ForcePhone` are Touch & Active [97] and PseudoButton [68]. Touch & Active attaches external contact piezoelectric microphones and speakers to objects to learn how the user is touching the object, which is achieved by fingerprinting the object's resonants. A follow-on study [98] showed that the same fingerprinting technique can also be used to infer the force applied to objects, which aimed mainly to ease product prototyping. Using the built-in speakers and microphones on commodity phones was mentioned as a potential interface for Touch & Active, but no further details were provided. Moreover, `ForcePhone` directly models and interprets the signal changes caused by the applied force, rather than classifying the resonants by using machine learning. The latter usually requires a significantly more effort for periodic training and retraining. On the other hand, PseudoButton utilizes similar sound degradation to estimate the applied force. However, it focuses on the sound degradation of airborne propagation, so it can only estimate the force changes at the microphone reception hole, e.g., the recorded volume decreases when the user completely blocks the microphone reception hole. In contrast, `ForcePhone` utilizes the structure-borne sound propagation, which can provide this force-sensitive touch around the entire body of each phone.

4.3 Structure-Borne Propagation

Sound is a mechanical wave broadcasted by compressions and rarefactions. The most common material for sound to propagate is the air, which is known as *airborne propagation*. Besides the air path, as shown in Fig. 4.3, when the sound is generated and received by



(a) Sent Sound Propagation (b) Received Sround Response

Figure 4.3: Structure-borne propagation in a phone. The sound received at a phone is a combination of structure- and air-borne propagations as well as the environments' reflections (echo). `ForcePhone` uses 20 samples before the strongest correlation peak as an indicator for a structure-borne propagation.

the same device, its body becomes another pathway for the sound to travel, which is called *structure-borne propagation*. Since the structure-borne propagation is usually unrelated to the intended application, it is generally considered as a noise [49, 84, 134]. However, `ForcePhone` utilizes it in a novel way to estimate the force applied to the phone by monitoring the degradation of structure-borne propagated sound.

As shown in Fig. 4.4, we model a vibrating phone as a forced, damped mass–spring system, where the phone vibrates up and down with the force F_v, which is caused by the sound played at the phone speaker. When there is no external force applied, the phone can vibrate freely with amplitude A_0, which is captured by the system's effective spring constant, K_0, and also the damping coefficient. Moreover, since the phone is considered as a rigid body, the equilibrium is located at A_0 above the table. Based on Hooke's law [66], when an external force F_h is applied to the screen, the system equilibrium moves downward by F_h/K_0. However, since the phone (a rigid body) is impossible to move downward "into" the table, the applied force also makes the system spring constant increase to K so as to meet this equilibrium change and rigid body constraint. In such a case, if the vibration energy is constant, the potential energy for both systems will be identical, thus leading to:

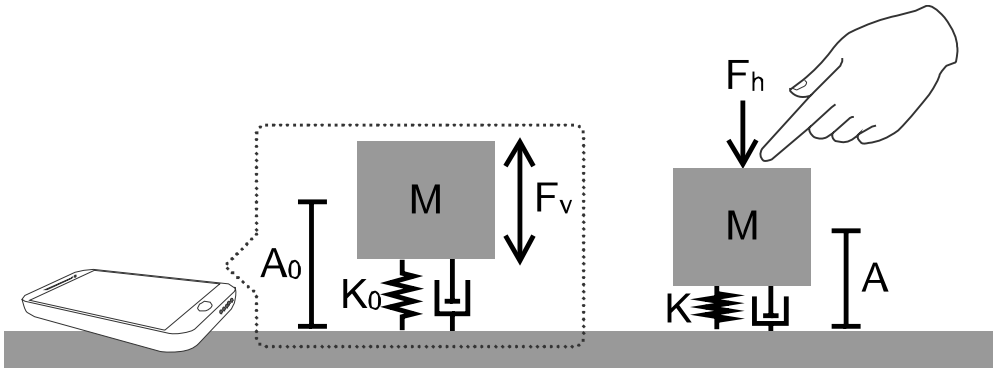
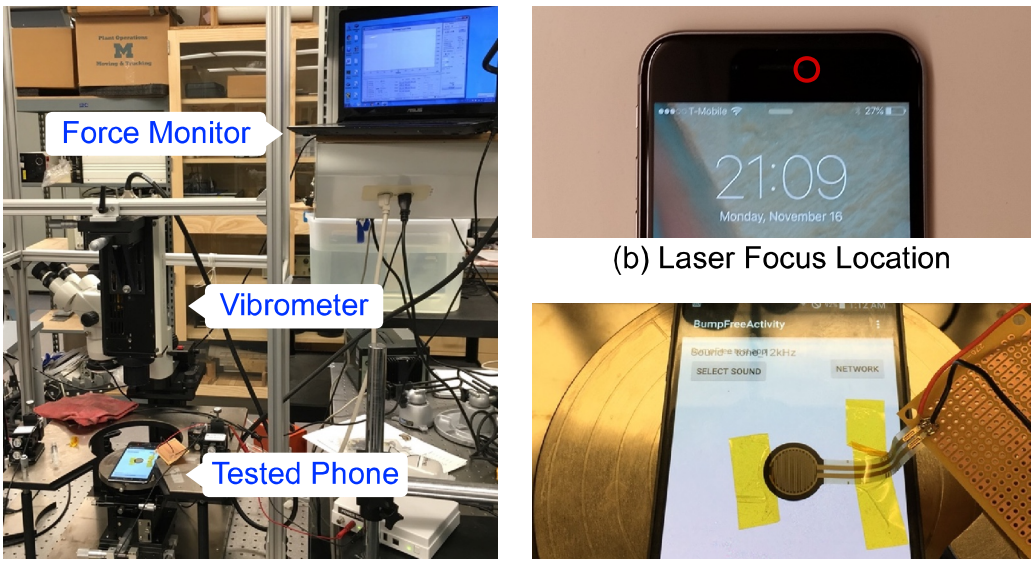


Figure 4.4: Phone vibration model. Phone vibration is modeled as a forced and damped mass–spring system where the phone vibrates with amplitude A_0 due to the force F_v from the phone speaker. The vibration amplitude is decreased to A and the effective system spring coefficient is increased to K due to the applied force F_h.

$$\frac{1}{2}K_0A_0^2 = \frac{1}{2}KA^2 = \frac{F_h}{A_0 - A}A^2 \tag{4.1}$$

which defines the relation between the applied force and the reduced vibration amplitude. Note that this basic model is designed only for an illustrative purpose as it doesn't consider the horizontal vibration and the increased friction caused by the applied force. Moreover, since the hand and the table also slightly vibrate with the phone, the system's damping coefficient and effective mass will change accordingly, which are not accounted for, either. However, according to our vibration measurements, this simplified model suffices to describe the principle of `ForcePhone`'s operation.

We used the Polytec OFV-303 laser vibrometer [26] to measure the nm-level vibration of a phone, capturing the change of structure-borne propagation caused by touching the phone with hand. Compared to capturing the structure-borne propagation via microphones (which `ForcePhone` is using as described in the next section), this vibration measurement helps neglect the noises caused by the air-propagated sound and the imperfect microphone hardware. Fig. 4.5 shows our experimental setting where the phone is placed on a metal surface under the laser vibrometer. The laser's focus has been calibrated before the test to ensure that the laser beam will be reflected properly by the phone surface. The latest

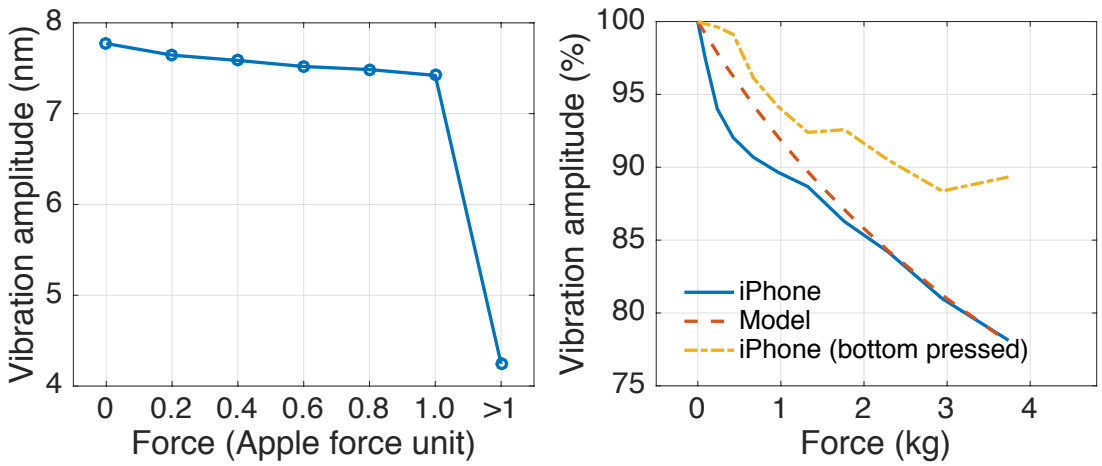


(a) Experiment Setting (b) Laser Focus Location (c) External Force Sensor

Figure 4.5: Vibration measurement setting. Vibration is measured by Polytec OFV-303 laser vibrometer when force is applied.

Apple iPhone 6s is used for these measurements owing to its capability of estimating the applied force. Thumb is used to apply force at the middle of the phone and the iPhone 3D Touch reading (ranging from 0 to 1) is used as a reference of the applied force. From our preliminary test, we found that the Apple 3D Touch sensors can only read the applied force up to about 380g. Hence, we also repeat the same measurements with the Interlink force sensor [16] which can measure force up to 10kg. Its current change caused by the applied force is recorded by the Monsoon power monitor [23].

Fig. 4.6 shows the results of phone vibration while applying different amounts of force. As shown in Fig. 4.6(a), when the Apple 3D Touch reaches its maximum sensitivity, the amplitude of phone vibration decreases about 5%. If more force is applied further (marked as > 1), near 50% of the vibration amplitude is damped by hand. Fig. 4.6(b) shows the results of measuring the force with external force sensors. The vibration amplitude also decreases about 5% when a 380g force is applied to the phone. Our system model is shown to be able to capture the main trend of vibration amplitude when the initial system spring constant $K_0 = 2.7$(kg/nm) is assumed. The most important property observed in this experiment is the high correlation between the applied force and the decreased vibration ampli-



(a) By Apple Sensors (b) By External Sensors

Figure 4.6: Phone vibration damped by force. The correlation between the damped vibration and the applied force enables `ForcePhone`'s force-sensitive and squeezable interfaces.

tude. Based on this property, `ForcePhone` can provide useful force-sensitive applications as introduced in the following sections.

As mentioned earlier, our model is not perfect. For example, as shown in Fig. 4.6(b), when the force is applied to the bottom of the phone, it incurs less vibration change since the measured location (i.e., the top of the phone) has less restriction when the phone's bottom is pressed. A similar phenomenon also occurs when different speakers are used to play sound. Thus, it is necessary for `ForcePhone` to make one-time calibration for unifying different estimations when force is applied at different locations on the screen. The calibration needs to be tuned to the phone models as they have varying vibration patterns depending on the phone material. For example, the vibration of Galaxy Note 4 is found to be 50% larger than the vibration of iPhone 6s, and more sensitive to the applied force since its plastic body is easier to compress. We also found, for some phone models, such as iPhone 6s, this vibration decays too fast when sound is played by the bottom speaker and received by the top microphone, making the force estimation less accurate. In such a case, we used the top speaker to get an adequate signal to noise ratio (SNR) of collected sound even though the collected sound will include a part of airborne propagation components.

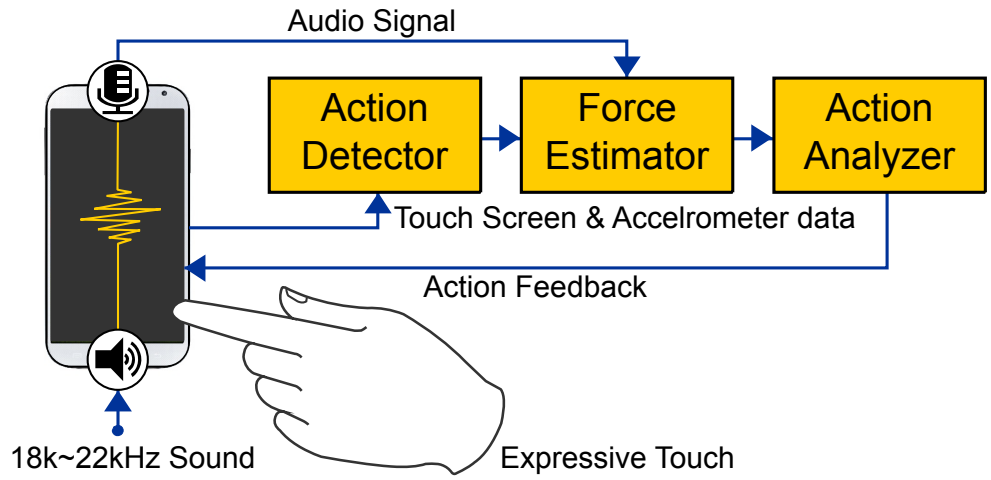


Figure 4.7: System overview. Force applied to the phone damps the inaudible sound sent from the phone's speaker to its microphone. Accelerometer and gyroscope readings are used to avoid other audio signal noises caused by movements.

4.4 System Design

As shown in Fig. 4.7, `ForcePhone` actively plays an *inaudible* sound with the phone speaker, and then picks up this sound with the phone microphone. The touch screen input (such as the location or the start time of a touch) and the data from the other motion sensors are also recorded. These sensor data are then used to improve the force estimation and reduce the number of false detections. When force is applied to the touch screen or the other parts of the phone body, the action analyzer triggers the pre-designed feedback/behavior based on the monitored (inaudible) sounds and user motions.

4.4.1 Sound Selection

The design of sound is critical to the system performance. While there are many other possible options, the current design of `ForcePhone` uses a 1200-sample linear chirp from 18kHz to 24kHz. A hamming window is multiplied to the first and last of 300 samples to eliminate the audible noise caused by spectral leakage [61]. The main frequency range of sound signals used to sense is about 20kHz–22kHz while the remaining signals close to 18kHz and 24kHz are played with minimal volume which are "stuffed" only to avoid

signal loss due to the windowing. ForcePhone samples this chirp at 48kHz and replays it every 50ms. This sound is designed to achieve (i) minimal user annoyance, (ii) high SNR, and (iii) adequate force sensing delay.

According to the field study in [102], humans are shown to have a limited ability to hear sound above 20kHz. Since most modern smartphones only support the sampling rate up to 48kHz, the highest sound frequency to use is 24kHz. This setting is likely to be expanded in the near future; for example, Android 6.0 has announced a plan to support a higher sample rate. Our experiments have shown that playing a high frequency chirp directly is still audible to humans because an abrupt increase/decrease of energy in time domain incurs frequency leakages. To avoid this problem, ForcePhone uses a similar windowing process as in [81] to reduce the noise burst at the beginning and end of signals. This windowing process adds an envelope to the sent signal that controls the signal amplitude to 0 in the beginning and then gradually restores the signal amplitude to the normal range (the same principle is also applied to the end of the signal). This design choice will cause SNR degradation when we estimate the audio correlation of received/sent signals, but it is practically important to avoid any audible sound. Note that the chirp starts at 18kHz but is still inaudible because the beginning of the chirp is played only at the minimal volume (windowing). None of the participants in our user study noticed/heard the sound used in ForcePhone.

Ideally, less stop time between chirps will provide a smaller sensing delay, but the chirps in ForcePhone are designed to be played every 50ms. This parameter setting is chosen to prevent the inclusion of audio signals reflected by environments as an unexpected/unwanted noise. For example, as shown in Fig. 4.3(b), after the sound transmitted via airborne propagation (i.e., the highest correlation peak) is received, there are multiple following local peaks which indicate the reception of sound reflected by the environment. From our experiments, the received reflections are found to degrade 20dB after 25ms, which is small enough to let ForcePhone play and receive the next round of chirp correctly with only

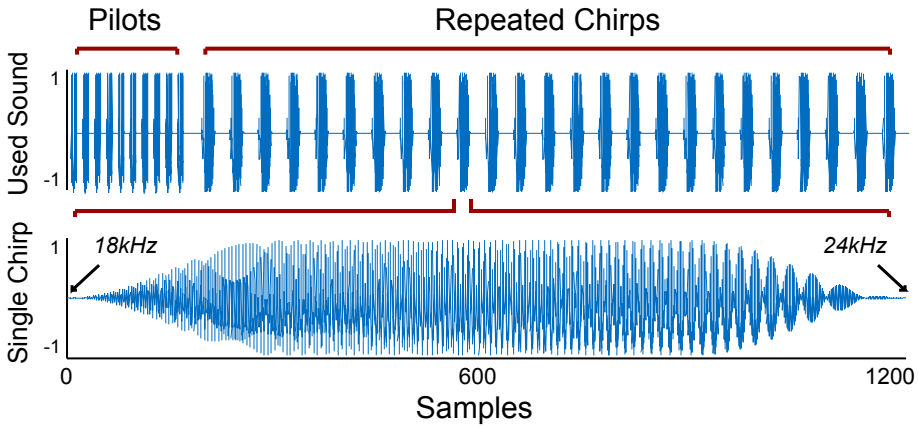


Figure 4.8: Example of transmitted inaudible sound. The pilots are used to synchronize the phone's microphone and speaker. The subsequent chirps stop for 25ms before playing the next chirp to avoid unexpected noises from environmental reflections. (This figure is rescaled for easy visualization)

minimal noise due to environmental reflections. Thus, it is necessary for `ForcePhone` to have a 25ms (i.e., 1200-sample) stop time after playing each chirp.

`ForcePhone` uses the signal correlation (also known as the *matched filter*) to estimate the reception of the played sound. The SNR of this correlation in the chirp is proportional to the signal length and sweeping frequency [112]. We selected the sweeping frequency as above in order to not annoy users and cope with the hardware limitation while setting the signal length to 25ms. Even though a longer chirp can achieve a higher SNR, it also increases the sensing delay because `ForcePhone` needs to wait for the completion of the sound being played. To strike a balance between SNR and the sensing delay, `ForcePhone` sets the chirp length to 25ms, which makes the total delay in sensing each chirp equal to 50ms (i.e., 20 force estimations can be made every second). This setting is found to provide enough SNR for estimation of the applied force and an adequate sensing delay which meets most users' needs.

Besides the design of sound signal, there is another parameter that affects the received SNR: the phone's speaker volume. In the current setting, the audio volume is set to 50% of the maximum to avoid some audible noise due to the imperfect speaker hardware and deal with the coexistence of multiple phones that run `ForcePhone`. An example of the

sent sound is shown in Fig. 4.8. One thing to note is that there are 10 additional chirps sweeping from 24kHz to 10kHz played before the above-mentioned chirps, which are used as the pilot signal for synchronizing the timing of the phone's microphones and speakers. If the synchronization fails, `ForcePhone` will stop the sensing process and replay the pilot to achieve correct time synchronization. Detailed performance measurements are presented and discussed in Section 4.6.

4.4.2 Estimation of Applied Force

`ForcePhone` utilizes the structure-borne sound propagation to estimate the applied force. As mentioned in Section 4.3, when force is applied to the phone body, the hand in contact with the phone body damps/degrades the structure-borne sound propagation. Besides the structure-borne propagation, there are other factors that affect the received sound strength. For example, the airborne propagation might be blocked by hand, and the overall sound signal strength may be enhanced by reflections from the environment or the internal resonant. In `ForcePhone`, these noises are identified and removed by timestamping the received audio signal. For example, the reflection from an object 10cm away will be received 28 samples later than the airborne propagation since it travels a 10cm longer distance. Moreover, sound usually travels 100x faster in a solid phone [74]. Thus, the structure-borne propagation will be received 21 samples ahead of the airborne propagation when the microphone and the speaker are 15cm apart. Based on these observations, `ForcePhone` uses the signal which is 20 samples ahead of the airborne propagation as the indicator of the structure-borne propagation, thus removing the most undesirable noise.

Note that the reference of airborne propagation is assumed to be the strongest audio correlation because the sound energy decays faster through the solid phone body and absorbed more on the reflection objects than air. Ideally, the temporal -3dB width of audio correlation are 7 samples for our chirp selection, so it would be possible to find a clear peak ahead of the airborne propagation, indicating the reception of structure-borne propagation. However, as

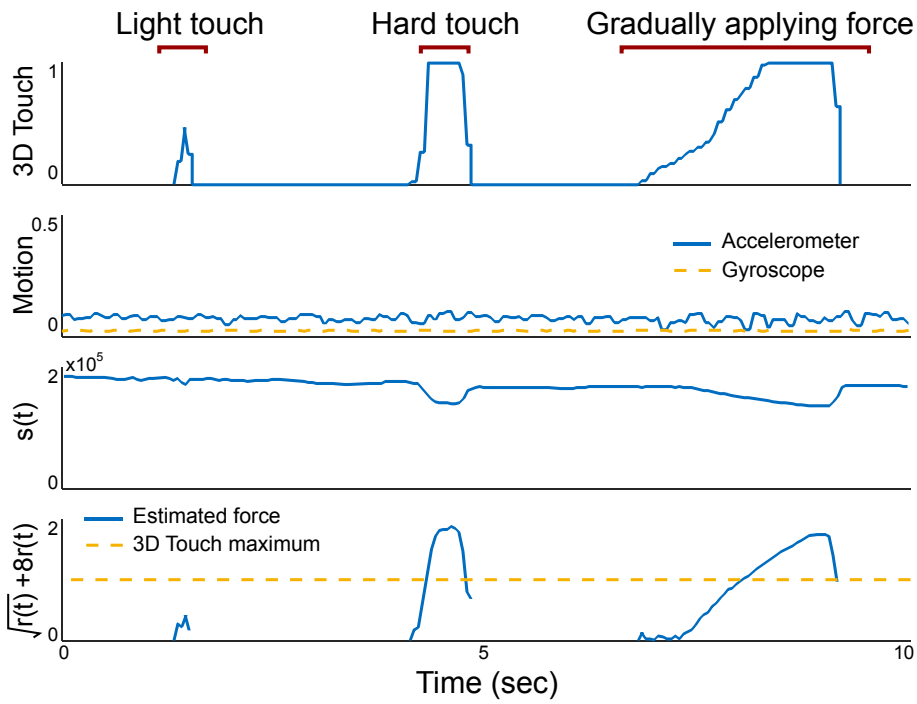
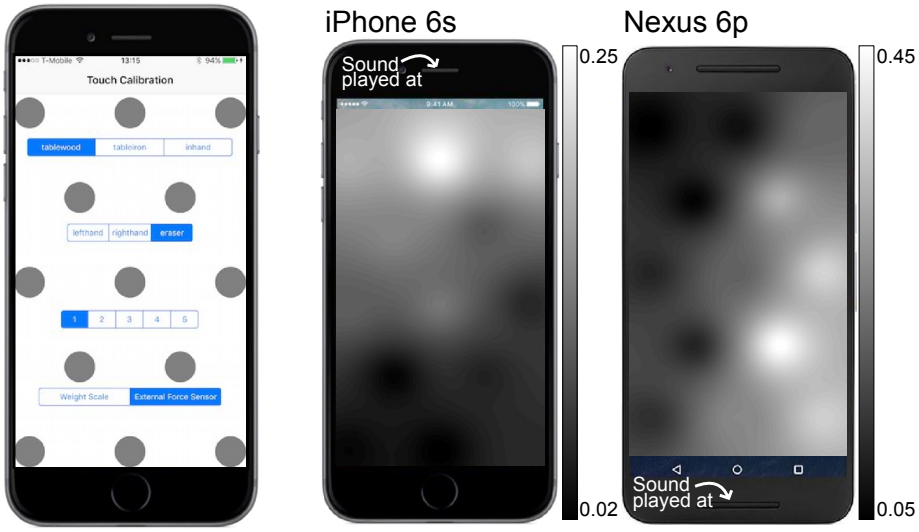


Figure 4.9: Responses of different amounts of applied force. Motion sensors only capture the initial response of a touch, but the sound response can monitor the subsequent applied force.

shown in Fig. 4.3(b), there is no such clear peak found before airborne propagation, and the temporal -3dB width of audio correlation is about 40 samples in this measurement. This phenomenon is caused by the adoption of windowing, which suppresses the frequency-domain signal leakage but incurs the time-domain signal leakage. This 20-sample-ahead sampling heuristic will thus include both air- and structure-borne propagations. To get a reliable estimation of the applied force, `ForcePhone` utilizes the sound strength when the touch begins as a reference to estimate the subsequent change caused by the force applied later. In the rest of this paper, we will denote the sound correction at time t as signal $s(t)$ and the signal at the beginning of a touch as s_{start}. The subsequent force at time t is estimated based on a metric called the *signal changing ratio*, $r(t) = |(s(t) - s_{start})/s_{start}|$. The applied force $f(t)$ at time t is then determined by a linear regression model with $r(t)$. Fig. 4.9 shows a real-world example of applying force to an iPhone 6s placed on a wooden table. We simply visualize the estimated force as $\sqrt{r(t)} + 8r(t)$; the intuition behind this



(a) Calibration UI (b) Signal change $\overline{r(t)}$ of 500g force

Figure 4.10: Touch calibration. The extent of signal changes caused by the applied force varies with the touch location. Thus, a one-time touch calibration is made at the 13 marked locations to compensate the estimated force at different locations.

setting will be discussed later. In this measurement, there are 3 different types of touch, each with a different applied force: (1) light touch, (2) hard touch, and (3) touch with a gradually increasing force. The ground truth of the applied force can be read from the Apple 3D Touch sensors [5]. As shown in the figure, motion sensors can only detect the slight movement at the start/end of a touch but are unable to determine the applied force. Our heuristic based on $r(t)$ captures most of the force characteristics even when the applied force exceeds the maximum sensing range of Apple 3D Touch.

Note the calibration between $f(t)$ and $r(t)$ varies with the location of force applied on the phone as described in Section 4.3. Thus, a preparatory experiment is needed for each phone model before `ForcePhone` is activated. However, this calibration is just a one-time requirement, which is different from other sound-fingerprinting systems that need laborious training each time before using the application.

Our current implementation of `ForcePhone` is calibrated by using external force sensors. The calibration is done by applying different amounts of force (up to 1.5kg) at 13 selected locations as shown in Fig 4.10(a). The remaining parts of touch screen are cali-

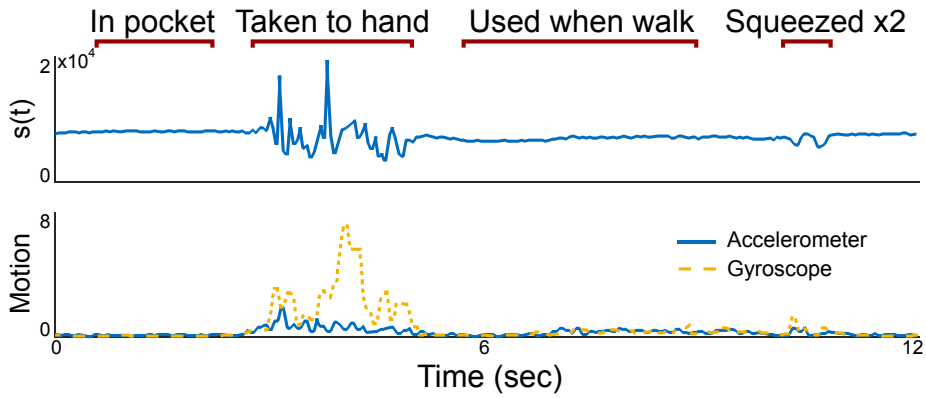
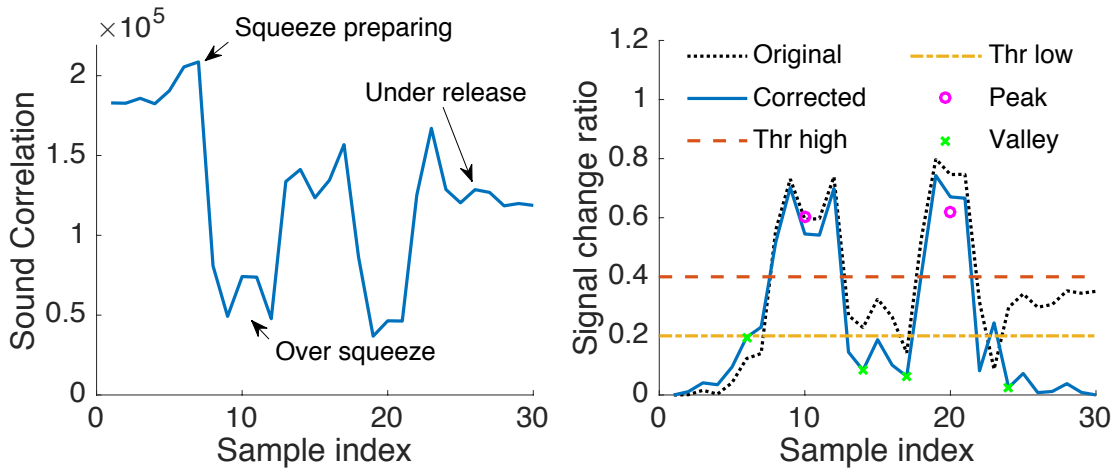


Figure 4.11: Response of movement and squeeze. Sound correlation changes when the environment changes, such as moving the phone from the pocket to hands, but it becomes stable quickly when people hold phones in their hands.

brated by interpolating the estimated force at those 13 locations. Fig. 4.10(b) shows examples of average signal change ratio, $\overline{r(t)}$, when a 500g force is applied to the 13 calibrated locations and the estimated $\overline{r(t)}$ over the remaining parts of the phone. As shown in this figure, $\overline{r(t)}$ varies with location due to different distances from/to the speaker/microphone and also with the structure of the phone. In general, touching near the speaker used to play sounds generates more pronounced signal changes while touching locations far away from the speaker causes less pronounced changes. Note that our current calibration is done when the phone is placed on a static surface, thus ignoring the airborne signal changes due to the movement of the phone during the touch. However, as we will discuss later, the added estimation noise due to the phone movement can be tolerated with proper app design since users are also unaware of the actual amount of force being applied to the phone unless the response from the phone is sensed. Most participants of our usability study felt comfortable with the current setting. Improving this calibration process with more advanced sensors and algorithms is part of our future work.

4.4.3 Squeeze Detection

It is challenging to detect the squeeze of phone body. Fig. 4.11 shows an example of the observed signal change when a phone is taken out of the pocket with hand, used while



(a) Squeeze response (b) Squeeze detection

Figure 4.12: Squeeze detection example. Received signal is first normalized by the start and the end of signal amplitudes. Peak is identified when the corrected signal passing the high threshold and the signal above the low threshold is considered as part of the peak.

walking, and then squeezed twice. As shown in this figure, even though the signal response of a squeeze is clearly observable, it is also possible to include lots of noise due to the phone's movements. To avoid this noise from large phone movements, we have built a motion detector based on both accelerometer and gyroscope readings, which turns off our squeeze detection when there is a large phone movement and the signal is noisy. We thus set a threshold to turn off the detection during large phone movements (such as taking the phone out of pocket and transferring it to a hand) but keep the squeeze function on during slow/small phone movements, such as walking and using the phone in hand.

Even though the movement noise is removed by the motion detector, determining if the user squeezes the phone is still harder than estimating the force applied to the touch screen. Due to the lack of touch screen input, we don't know when the squeeze starts. Moreover, the location of squeeze is inaccessible, thus making it difficult to perform a proper calibration. To solve these issues, our squeezable back function is made to respond only to a predefined squeeze behavior, such as double squeezes applied to the phone's left and right body frame as shown in Fig. 4.2, and the entire squeeze process is assumed to complete in 1.5 seconds.

Fig. 4.12(a) shows an example of this predefined squeeze response, where the applied squeezes cause two significant signal drops at time 10 and 20. Besides the signal response to the squeeze operation, there are three characteristics of human squeezing behavior, which are critical to our design of squeeze detection. First, the the applied force might decrease slightly before a squeeze, probably because the users need to relax their hands before squeezing their phone. After the squeeze, the applied force may also decrease around the expected force peak. We call this phenomenon *over-squeeze*, and think it is caused by the fact that users actually release part of their fingers/palms from the phone body when they try to apply more force with the other part(s) of the hand. At the end of squeeze, the applied force may not return to its initial condition either, because the users may change their position of holding the phone during the squeeze.

Squeezes are also detected based on the signal change ratio $r(t)$. `ForcePhone` continuously checks each 30-sample received signal to see if the predefined squeeze pattern has occurred. As the hand position is likely to change during a squeeze, it is less accurate to estimate the applied force based only on s_{start}. For example, the larger drop of the second squeeze shown in Fig. 4.12(a) might be caused by a hand position change rather than a larger squeezing force. To account for this phenomenon, we heuristically set the reference signal based on both the start and end of this 30-sample signal. This new reference signal varies over time and is defined as $s(t)_{reference} = ((t_{end} - t)s_{start} + (t - t_{start})s_{end})/(t_{end} - t_{start})$. The signal change ratio $r(t)$ is calculated based on $s(t)_{reference}$ using the same process as described earlier. Basically, this method estimates $r(t)$ by weighting the reference signal proportional to the difference between t and t_{start}, t_{end}. Fig. 4.12(b) shows the result of this correction. After estimating $r(t)$, `ForcePhone` sets two thresholds to identify the force peaks caused by squeezes. A peak occurs when $r(t) > thr_{high}$. The following samples where $r(t) > thr_{low}$ are defined as parts of this identified peak. This setting is designed to avoid false identification due to the *over squeeze* and *squeeze preparation*. Thus, thr_{high} is set larger than the assumed squeeze preparation force change, and thr_{low} is set smaller

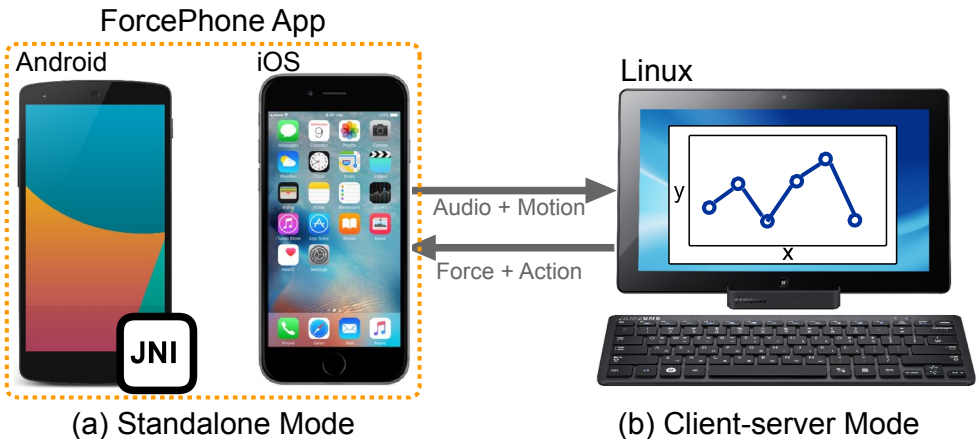


Figure 4.13: Implementation overview. `ForcePhone` has been implemented as a standalone app on Android via Java Native Interface (JNI). Our iOS implementation requires the force estimation done at a remote server.

than the over squeeze force change. A valid squeeze is identified when the number of detected peaks, the peak widths, and the peak-to-valley ratios fit a predefined criterion. The performance of our current setting will be evaluated in Section 4.6.

4.5 Implementation

We have implemented `ForcePhone` on both Android and iOS devices. As shown in Fig. 4.13, our Android implementation runs as a standalone app; the force estimation is implemented as a library in native code (C++) which we integrate into the app via Java Native Interface (JNI). On the other hand, the current iOS implementation requires the force estimation to be done at a remote server using Matlab. Both implementations provide real-time force estimations. The delay between recording each chirp and receiving a force estimation is 15ms and 61ms at our local and remote implementations, respectively. We are currently porting the force estimation library implemented in C++ to iOS.

One implementation challenge is that `ForcePhone` needs accurate time synchronization between the phone speaker and microphone to extract valid structure-borne propagations. However, both iOS and Android platforms don't meet this real-time requirement since both take 20–100ms to run the play/record audio command. To overcome this difficulty,

ForcePhone adds 10 pilot sequences ahead of the used chirps. In this synchronization, ForcePhone checks if the received signal contains the identical sent pilots, such as having the same stop time among pilots. In our experiments, more than 95% of the received pilots have less than 5-sample jitters in their stop time, and 15% of trials on Android experience a special 960-sample delay before the 4-th pilots. The criterion for ForcePhone to finish this synchronization test is the existence of these 10 pilots, and 50% of received pilots have the exact same stop time, and the last three pilots have less than 5-sample jitters. Once the pilots are identified correctly, the timing of the last pilot is used as a reference to extract the structure-borne propagation from the subsequent chirps as described earlier.

4.6 Evaluation

We first measure system performance, such as force estimation accuracy, overhead, and robustness to noise. We then measure the user's benefit of using ForcePhone, such as the probability of an assigned request being honored or whether the users think it useful.

4.6.1 Accuracy of Force Estimation

We evaluate the accuracy of force estimation by using the mean square errors and correlation coefficients where the former represents the ability to estimate exact force while the later indicates the capability to learn how force is changed. Fig. 4.14 shows an example of 12 different touches when the iPhone 6s is held in left hand and the center of the touch screen is pressed with left thumb. The applied force is estimated by the internal Apple 3D Touch sensors, external force sensors, and ForcePhone. Since Apple 3D Touch can only provide normalized results and no documented interpretation of the sensed values is available, we built our own post-hoc translation based on an external force scale. We found the sensed value is linear in the applied force, and the maximum value is reached when 380g is applied.

As shown in Fig. 4.14, the proprietary sensors used in Apple 3D Touch captures the

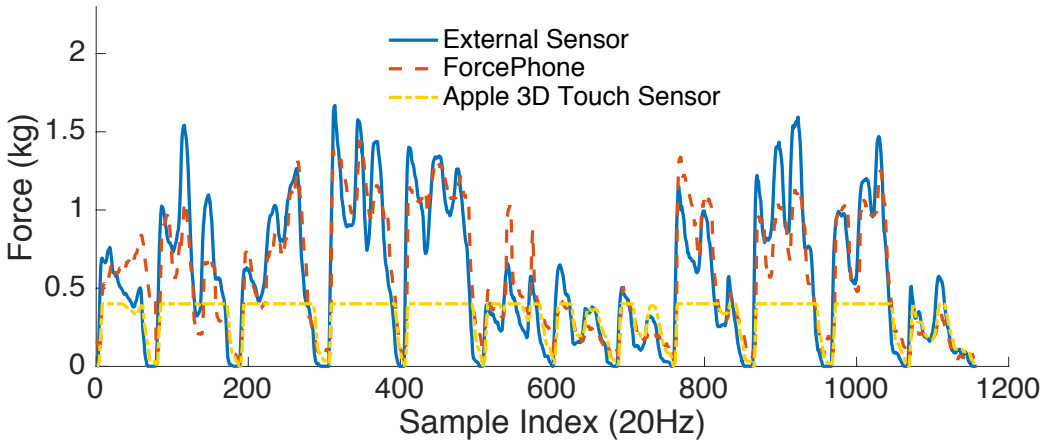


Figure 4.14: Accuracy of force estimation. 12 touch events with different amounts of applied force are plotted. The force estimated by `ForcePhone` provides high correlation with the ground truth estimated by using our external force sensors.

force change below its maximum (380g in this case) with high accuracy. Even though `ForcePhone` estimates the force under 380g with less accuracy, it captures most force-change characteristics as shown in Fig. 4.14. Meanwhile, `ForcePhone` can estimate the force above 380g without using any additional sensors.

The mean square error of this in-hand example is 205g and the correlation coefficient to the force estimated by external sensors is 0.87. Compared to the maximum sensed force up to 1.5kg, this error is tolerable for most force-sensitive apps as shown later. When the phone is stationary on a wooden table, the mean square error falls to 54g and the correlation coefficient increases to 0.91. Note that the error in estimating the exact value of the applied force could change if the force is applied in a different way than our calibration (as we don't consider the damping coefficient change in our current model as introduced in Section 4.3.) For example, there might be an estimation drift, so applying a varying force from 500g to 1000g might be estimated wrongly as changing from 400g to 900g plus the previously-mentioned errors. However, this feature doesn't hurt the experience of using `ForcePhone` because users are not aware of the exact value of the force applied to a phone unless this value is shown in the user interface [106]. With a proper user interface design, even though the estimated force is 100g less than the real force applied to the phone, users

can easily learn to adjust the applied force for getting a correct response. Actually, even the proprietary sensors used for 3D Touch have about 100g errors between the corners of touch screen, and it is suggested not to be used for estimating exact force [3]. In our current setting, when the force is applied to the middle area of the phone, the estimated force (in g) is calculated as $f(t) = 15 + 230\sqrt{(r(t))} + 2800r(t)$ in iPhone 6s and $f(t) = 1900r(t)$ on Galaxy Note 4. As shown in our usability study, this setting is adequate for building useful force-sensitive and squeezable apps. Calibration while considering how users touch the phone is part of our future work, which could be accomplished by other grip detection systems like GripSense [57].

4.6.2 Noise and Interference

The estimation of force based on sound propagation becomes erroneous if there is a strong audio noise with a similar structure as the chirps used in `ForcePhone`. For example, the mean square error of `ForcePhone` can increase from 50g to more than 500g when the same chirps are played by nearby computers's speakers. Even tough this is the natural limitation of any audio-based application, it is not the case in most real-life scenarios. In our measurements, `ForcePhone` is shown to be resistant to: (1) real-life background audio noise such as music being played in the testing room, (2) inter-device interference from `ForcePhone` running on another nearby device, and (3) self-interference from the audio played on the same device. To study the performance degradation caused only by the audio noise, rather than the force instability of human hands or other movement noises, we kept the test phone stationary on a wooden table and no force is applied. The standard deviation of force estimations is recorded and reported. This estimated force variation also represents errors when there is a force applied to the phone if the received noise is modeled as an additive noise.

Particularly, we use an Apple MAC Air or an iPhone 6s placed on the same surface as test devices. Multiple sets of noises were played, but no significant differences were

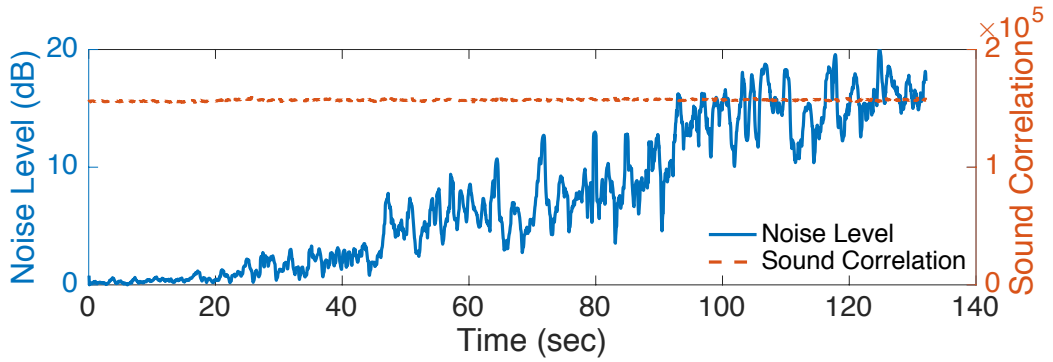
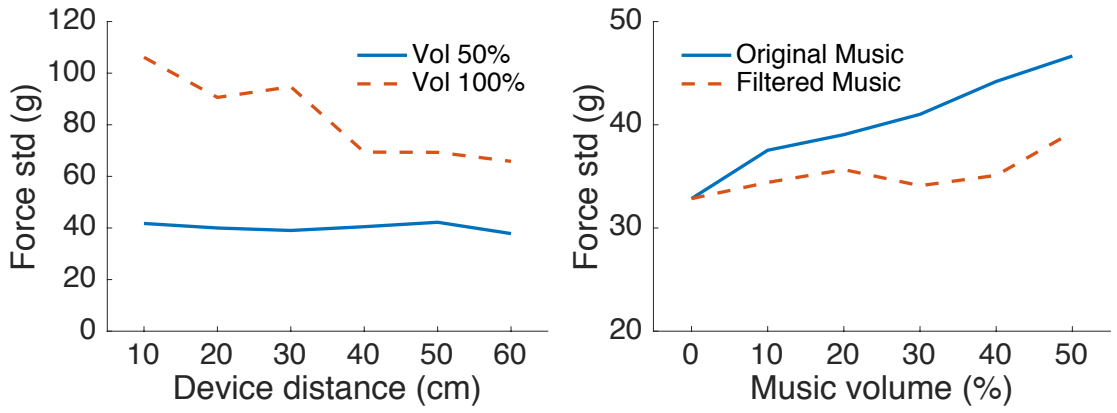


Figure 4.15: Resistance to background noise. Music (i.e., noise) played by a laptop 20cm away from the device under test has limited effect on the sound correlation even if the noise level is increased to 20dB higher than the used chirps.

observed, so we only report the results of music-playing noise, i.e., rock music "Jump – Van Halen" or gaming background music "Spot On". Since we place both the test device and the noise source on the same table, the results account for the vibration coupling and the noise transmitted through the surface.

We found `ForcePhone` to be resistant to most external real-life noises. As shown in Fig. 4.15, even when the rock music is played at full-volume by a laptop 20cm away from the test device, the sound correlation of the used chirps doesn't change much. This is because most human (singing or chatting) noises have limited signals in the 18kHz – 24kHz range. Moreover, this natural noise usually has only a limited correlation to our 1200-sample chirps. In this measurement, the increased estimation errors are less than 10g, which does not affect most of `ForcePhone`'s functionality, even when the noise level is 20dB higher than the used chirps.

Compared to the uncorrelated noises, `ForcePhone` is more sensitive to the noise that has a similar structure as the chirp it uses. Solutions of this issue are critical in allowing multiple phones to run `ForcePhone` in the same environment. To measure this, we had another device playing the same chirps used by `ForcePhone`, but with random stop time between consecutive chirps. This is designed to learn the average performance when two devices run `ForcePhone` simultaneously. As shown in Fig. 4.16(a), the average estimated error increases to 110g when the correlated noise is played at full-volume by another de-



(a) Inter-device interference

(b) Self interference

Figure 4.16: Resistance to inter-device and self interferences. The variation of sound correlation for each second is used to indicate the error when another device is running `ForcePhone` or a music is played on the same device.

vice. This trend is mitigated when the noise volume is reduced. Since `ForcePhone` only requires the designed chirps to be played at 50% of full-volume, multiple devices can run `ForcePhone` if they are at least 10cm apart from each other.

In the last scenario, we evaluate whether other audio signals can be played on the same device when `ForcePhone` is running. This capability is critical to the user experience, especially when using `ForcePhone` for a game control. We play a game background music from the same device when `ForcePhone` is running, and record the increased estimation error. As shown in Fig. 4.16(b), when the music is played by the same speaker at 50% of full-volume, `ForcePhone` experiences about 46g of additional estimation error, which does not affect the core functionality of `ForcePhone`. Moreover, when the played music is processed further by using a 15kHz low-pass filter, the imposed error decreased further to 37g. Note that the current design of `ForcePhone` allows game music to be played at most 50% of full-volume, as the other 50% is dedicated for `ForcePhone`'s functionality. This limitation can be addressed when the phone development kit supports streaming music/chirps to different speakers (devices usually have 2 or more speakers, but only one of them can be used at a time).

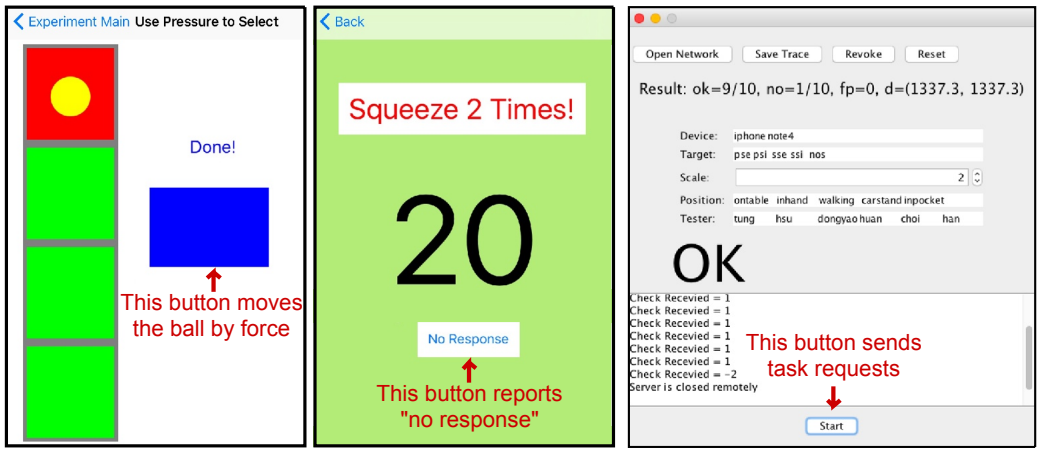
Setting	Idle	Backlight	ForcePhone	Website	Game
Power (mean)	33	856	1160	2564	3692

Table 4.2: Power consumption (mW). The additional power consumption by `ForcePhone` is about 304mW, which is small relative to that of normal phone usage.

4.6.3 Power Consumption

We measure the power consumption of `ForcePhone` with the Monsoon Power Monitor [23] on Galaxy Note 4. For the baseline measurements, we turn off the phone's radios and estimate the force through our JNI/C++ local implementation. We ran each scenario for 20 seconds in 6 different states: 1) idle, with the screen off, 2) backlight, with the screen displaying a white background, 3) `ForcePhone`, with the screen on, 4) website surfing, when networking is enabled, and 5) video game, "Puzzle&Dragon". These measurement results are summarized in Table 4.2.

Activating the naive implementation of `ForcePhone` consumes an additional power of 304mW compared to the case when nothing is executed and background is on. More than 90% of this power consumption comes from using the hardware of speaker, microphone, and accelerometer — this is consistent with the results in [39]. The computation cost of `ForcePhone` is minimal since it builds a model to estimate the applied force rather than fingerprinting the sound changes. Note that the ratio of `ForcePhone`'s power consumption is dependent on its usage. For example, though `ForcePhone` incurs 26% power overhead (compared to the blacklight case) when it is used to select app options at home screen, users won't press this option for a long time and `ForcePhone` can be turned off once the selection is done. If the proposed squeezable back function is used for web surfing, activating `ForcePhone` incurs only an 11% power-consumption overhead, which reduces the life of Note 4's 12 Wh battery by about 30min. The percentage overhead is reduced further if `ForcePhone` is used for game control or when microphones or speakers have been activated for other purposes.



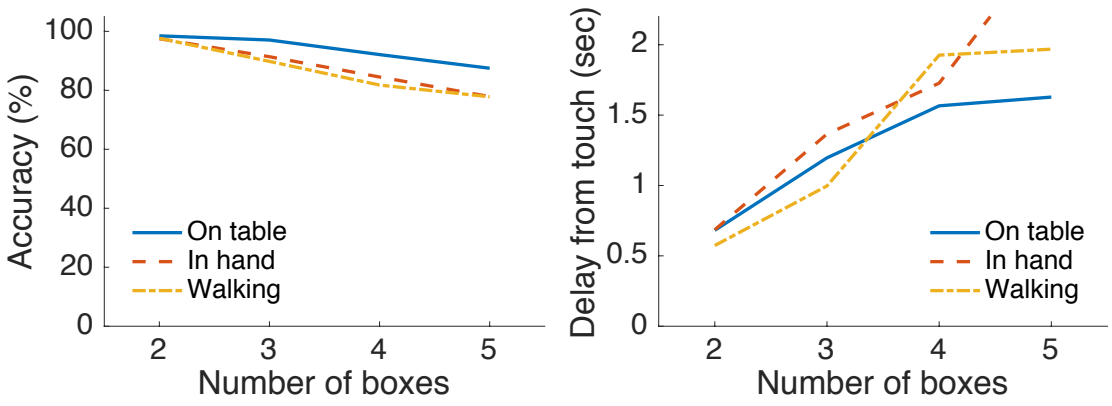
(a) Experiment App (b) Experiment Controller

Figure 4.17: User interface for experiments. Users are requested to move a ball to the marked red box by applying different amounts of force to the blue button and squeeze the phone twice for surfing the previous web page. Action requests are sent, and the results are recorded by the controller.

4.6.4 Usability Test

We recruited 6 participants (4 males and 2 females) to test the usability of `ForcePhone` by asking them to perform a set of tasks as shown in Fig. 4.17(a). Since these 6 participants are students who are aware of this project, we only tested their ability to complete the assigned tasks. (The results of the other users' opinions on `ForcePhone` will be provided in the next subsection.) In the first task, users are asked to move a ball by applying force and stop it at a randomly-selected red box; the highest box is reached by applying a 500g force and the ball located at the bottom of boxes (0g) when the task starts. This experiment follows a similar design principle as that in [106] to test if the user can effectively control the force at different levels. Experiments start from stopping the ball at one of the two big boxes (i.e., only two levels) to one of the five small boxes. We asked users to perform these tasks at different positions, such as controlling the phone while it is on a table, while holding it in hand, or while walking and using it simultaneously. A remote controller sends requests and records the accuracy and delay of users' reaction as shown in Fig. 4.17(b).

Fig. 4.18 shows the accuracy (the success rate of moving the ball into the selected box) and the delay between the user touching the blue button and finishing the task. The plots

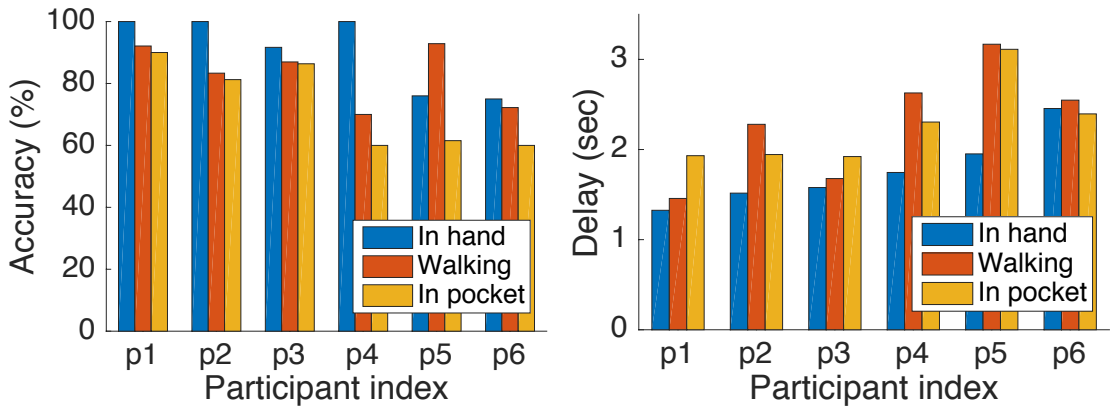


(a) Accuracy (b) Delay

Figure 4.18: Result of controlling a ball with ForcePhone. Results are averaged over 6 participants. Delay is estimated as the time between the user presses/releases the button.

show that most participants can effectively control ball movement with `ForcePhone` when there are only 2 – 3 boxes. This result supports our hard-pressed option app since the users can easily control the applied force at two levels with higher than 97% accuracy. When more than 3 boxes (levels) are provided, users can still achieve higher than 90% accuracy when the phone is stationary on the table. This is consistent with our earlier evaluation of estimation accuracy. Although different touch positions might incur additional errors, the high correlation between the force estimated by `ForcePhone` and the real force is sufficient for users to control the phone effectively. It is worth noting that moving the ball toward the top/bottom boxes is easier than the box in the middle; moving the ball to the bottom or top box has 13% higher accuracy than moving the ball into the middle box. A similar property was also reported in [106].

On average, users require only 0.7 second to stop the ball at the correct position when there are two boxes. This delay increases to 2 seconds when the users attempt moving the ball among 5 boxes while walking. Interestingly, controlling the ball when the phone is stationary is considered the easiest task, but the participants use a slightly less time to complete the 2-box task while walking. We conjecture this phenomenon is caused by the order of tests we perform, where the walking test is executed after testing the phone–table and in-hand cases. This property implies that performance can be improved further once



(a) Accuracy (b) Delay

Figure 4.19: Results of squeeze detection. The accuracy of the last three participants is increased to more than 90% when the detection criterion is adjusted after this test.

the users become familiar with `ForcePhone`. The average accuracy of repeating the same task on the Galaxy Note 4 is increased by 3%. We think this minimal improvement is due to the users' familiarity with this task rather than the force estimation accuracy. In summary, users can effectively use `ForcePhone` to control different levels of applied force for various scenarios.

In the second experiment, we asked the participants to squeeze the phone's body. Once `ForcePhone` detects the squeeze, the testing app switches to the previous UI page with a different background and indicates the page number as shown in Fig. 4.17(a). Due to the lack of ground truth (no touch event exists), we asked the users to click the *no response* button once the user observes the applied squeeze not detected. We count the number of times the squeeze is detected without any instruction from the remote controller as a false positive.

Fig. 4.19 shows the accuracy and delay for each user. Participant 1 has the highest accuracy and the lowest delay because the detection setting is tuned according to his own preliminary test and since he practiced this task more than the other users. The average accuracy when the phone is held in hand is 90%. The accuracy drops to 82% when the participants use their phones while walking. Due to the lack of timing information on when the user applies and stops the squeeze, we only record the delay between the instruction and

the squeeze feedback (either success or fail). As shown in Fig. 4.19(b), the average delay of each squeeze is about 2 seconds. Users experience low accuracy usually experience a long delay since they have to wait and then click the "no response" button if the squeeze is not detected. From the measurements of our previous moving-ball test, we found users usually need 0.8 second to react to the displayed instruction, and hence the delay of squeeze detection is expected no longer than 1.5 seconds.

It is possible to detect a squeeze in users' pocket, which can be used to turn off alarms or notifications when the phone is in pocket. However, our measurement shows that its performance depends not only on how the phone is squeezed but also on the clothing material. Tight jeans (e.g., worn by p1 to p3) generally have a better detection accuracy than loose pants. During this controlled experiment, only 4 false detections are observed. Users' feedback on the false positive rate will be presented in the next section.

Based on more than 600 squeeze profiles collected from the 6 participants, we tuned our final squeeze detection setting as follows. We set the high and low thresholds of squeeze detection to 310g and 175g, respectively. We identify signals as a peak only when the peak width lies between 2 and 8 samples and the peak-to-valley ratio is greater than 2.5. By applying these new criteria to the collected traces, the overall detection accuracies for the last three participants increase to higher than 90% for both in-hand and walking scenarios. We used the same setting in our subsequent application tests for other participants.

4.6.5 Users Study of Proposed Apps

For a users study of `ForcePhone`-based apps, we randomly chose 21 participants (7 females and 14 males) among a large student population at the University of Michigan. All of recruited participants own smartphones; only 3 of them have the latest iPhone 6s and knew how to use 3D Touch prior to this test. In contrast to our previous usability test, these participants were unaware of `ForcePhone` and our research group (so they have no initial bias). We first ask the participants to use the built-in hard-pressed option of

Questions	Strongly disagree /Disagree	No opinion	Strongly agree /Agree
Hard-pressed option is helpful	0	0	21
Hard-pressing(3D Touch) is responsive	1	1	19
Hard-pressing is responsive	0	1	20
Ball-moving game is interesting	0	3	18
Moving ball is responsive	2	4	15
Squeezable back is helpful	1	2	18
Squeezing is responsive	2	1	18
False detection is acceptable	4	7	10

Table 4.3: Application study results. Survey questions are given after users try the hard-pressed option, ball-moving game, and squeezable back applications for 10 to 15mins.

iPhone 6s and then try the same function implemented by `ForcePhone` on Galaxy Note 4. We built a fake UI (as shown in Fig. 4.2(a)) to make them feel if they were triggering the real option on Android. We set the testing threshold for enabling options to 340g, which is similar to the iPhone's. We chose not to make a blind test by using our iPhone implementation because the current iOS disables the vibration service when audio is being recorded. According to our preliminary test, vibrating the phone when the hard-pressed option is being activated is critical to the user experience. After trying the hard-pressed option, the users were instructed to play a simple game with the moving-ball test app. This allows us to record the users' feedback on using `ForcePhone` as a continuous UI input. While playing the game, users have to move the ball into the red box, and the number of boxes was increased when the users completed the task. Last, we asked the participants to test our squeezable back app that automatically navigates the previous UI page when they squeeze the phone twice. The threshold is set based on the results of our previous usability study. It took about 15 minutes on average, to complete the entire test.

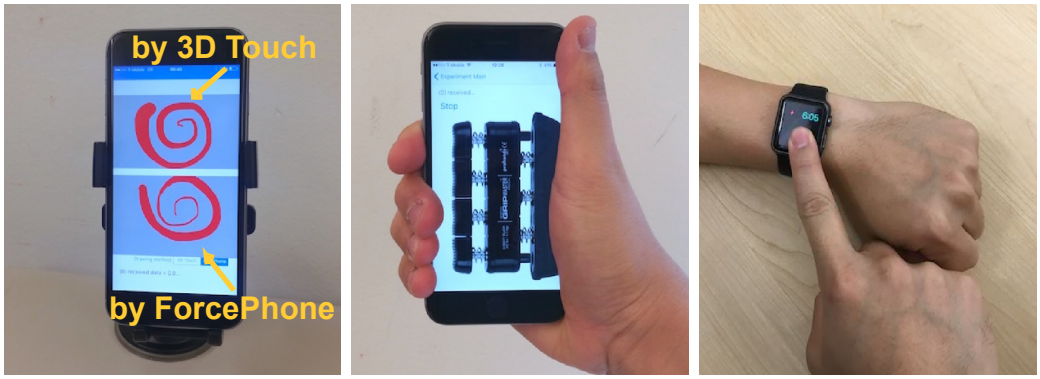
After testing our app, a survey form was filled out by the participants, and the results are summarized in Table 4.3. The question mark by "3D Touch" in Table 4.3 refers to the hard-pressed option implemented by Apple 3D Touch, while the others refer to `ForcePhone`. Most users are positive of the proposed apps and think them helpful. For the hard-press

option, most users think ForcePhone has a comparable performance to Apple 3D Touch. One of them said ForcePhone is better than 3D Touch because the vibration in Android is much clearer (stronger) than iPhone's, which is not related to the force detection. Only 3 users think iPhone's performance is better, but still acknowledge ForcePhone is responsive enough for the hard-pressed option app. This indicates that ForcePhone can handle simple tasks with a comparable performance as adding proprietary sensors. Moreover, most users feel the squeezable back app is helpful, which is a unique capability of ForcePhone.

Most users regard that controlling the ball based on the applied force is relatively difficult, but still were able to control the ball. Two users think our test setting is too sensitive, making it difficult to move the ball. We also discovered some errors caused by applying a large initial force and releasing the button immediately, which was not the intended case for ForcePhone. After the users are instructed to move the ball by gradually applying force, they are able to control the ball with ForcePhone. The squeezable back app received a similar rating as the ball-moving game. In our other survey, 16 users indicated difficulty in clicking the app back button when operating the phone with one hand which supports the design of our squeezable back app. Most users think our current parameter setting tuned by previous 6 participants is responsive and acceptable. A half of users experienced false detections when they moved the phone from one hand to the other. During the test, none of users heard the sound used in ForcePhone, so no user annoyance. The test locations were close to a cafe crowded with students, but ForcePhone was robust to human noises. Most common comments from the users are "cool idea" and "useful'. We plan to have extensive and large-scale tests before releasing ForcePhone to public.

4.7 Discussion

ForcePhone has been shown to be able to expand user input interfaces by using only built-in sensors. Several demonstrative apps have been developed and tested, but there are many other use-cases of ForcePhone. Discussed below are the limitations of current



(a) Drawing app (b) Hand-trainer app (c) Wearable app

Figure 4.20: Potential usage of ForcePhone.

ForcePhone and some of possible directions of our future work.

4.7.1 Limitations

As our evaluation results show, ForcePhone's force estimation includes noise from object contacts and human movements, calling for an app design to eliminate this noise. An extreme example that breaks ForcePhone's functionality is to activate the force estimation by placing the phone on a table then quickly moving and holding it in hand. However, this limitation of ForcePhone can be avoided by a proper app design. For example, the hard-pressed option app only needs force measurements within a short period of time (e.g., 2 seconds) and is inactivated if the phone is being moved. Apps like drawing lines with different line styles/thickness by applying different amounts of force is not suitable for the current ForcePhone as it is more probable for the user to (unintentionally) change the environment during the line drawing. But selection of thickness by applying different amounts of force is possible since it is akin to our ball-moving test. In our measurement, to achieve accurate and long-lasting force sensing, ForcePhone needs the phone to be fixed at a certain location. For example, Fig. 4.20(a) demonstrates the above-mentioned force-sensitive drawing app by attaching the phone on a stand.

In our squeezable back app, missed detections occur when the detection is intentionally turned off upon identification of extensive movements, such as sudden turns or stops. This

is a safety mechanism for ForcePhone to avoid false detections, but the users are not aware of this. One way to avoid this misunderstanding and improve the user experience is to provide a feedback when the squeeze detection is temporarily turned off. For example, the background of website or the title bar can be dimmed slightly when the squeezable back app is deactivated. Studying the users' reaction to this new UI design is part of our future work.

While the touch location near the speaker/microphone used to play/record sounds is found more sensitive to the applied force (i.e., making more pronounced signal changes), the middle area of touch screen usually provides more reliable force estimation. For example, when more than 1.2kg force is applied to the top area of iPhone 6s, the received signals jump quickly and the relation between $f(t)$ and $r(t)$ breaks down. We suspect this phenomenon is due to the pressuring of phone's microphone and other internal components. So, we suggest to limit the estimated force to 500g for consistent performance, even though certain locations can sense the force up to 3kg. This phenomenon also causes the estimated signal strength $s(t)$ at certain locations to rise when the force is applied. Most of these situations caused by pressing the phone can be compensated by our calibration of $r(t)$ and the estimation constraints caused by this problem will likely be addressed in the near future since commodity phones start to increase the number and fidelity of speakers/microphones. For example, when multiple speaker–microphone pairs are used to sense force interactively, the locations yielding erroneous estimations for a single microphone–speaker pair can be corrected by the other microphones and speakers. We should also note that, as shown in our measurements, these minor issues incur minimal annoyance to users when ForcePhone is properly applied.

4.7.2 Potential Applications

The participants of our usability study suggested many potential uses of ForcePhone after trying the proposed apps. Of them, two most promising and interesting uses are

mobile health apps and force-sensitive wearable devices.

Mobile phones or wearables are good candidate platforms for mobile health apps since people carry them all time. For example, systems of notifying/requesting users to walk or excise are now built in most commodity phones [4, 28]. Researchers have also shown the benefits of drinking more water via mobile apps [45]. `ForcePhone` can provide a new function/capability of these systems because it can determine how hard a user squeezes/touches the phone. As shown in Fig. 4.20(b), this functionality can be used as a replacement of force-finger-trainer that is helpful for the people with disability and those who use hands and fingers excessively for their work, e.g., computer programmers.

`ForcePhone` can also be implemented in wearable devices that have even less space for user inputs than smartphones. Unfortunately, current wearable devices usually have less sensing capabilities than smartphones, especially for the microphone sensitivity and sample rate. For example, Samsung Gear S (watch) can only support an audio sample rate up to 32kHz, which is inadequate for the current design of `ForcePhone`. Apple Watch hardware is found to be the best to implement `ForcePhone`, but its current API does not yet support data reading from the audio queue and process audio data in real time. In future, we expect the sensing capability of wearables to improve so that `ForcePhone` can be implemented and used for numerous apps.

4.8 Conclusion

We have proposed `ForcePhone`, an inexpensive solution that adds a force-sensitive interface to commodity phones without additional hardware. `ForcePhone` has been implemented on iOS and Android, and several apps based on its functionality have been developed and tested. Our evaluation has shown `ForcePhone` to provide comparable performance as augmented proprietary force sensors and to be robust to most real-life noises. The proposed apps are easy to use with higher than 90% accuracy and minimal overheads. In future, we plan to test `ForcePhone` on a wide range of devices and scenarios.

CHAPTER V

LibAcousticSensing

5.1 Introduction

Over the past decade, acoustic sensing has drawn significant attention thanks to its ubiquitous sensing capability. For example, it can be easily installed on numerous existing platforms, such as laptops, smartphones, wearables, or even IoT devices, because most of them are already equipped with microphones and speakers. Acoustic sensing is also versatile; it can provide context-aware computations [37, 76, 83, 88, 96, 109, 118], extend human–computer interfaces [47, 58, 62, 79, 94, 119, 124], and create nearly zero-cost inter-device interactions [36, 56, 81, 92, 115, 131, 132]. Even though these applications are designed to be *ubiquitous*, most of them are implemented only on a single platform like Android or iOS, and tested with just one or two types of devices.

Different platforms/devices, in general, have significantly different sensing capabilities due to their varying hardware and operating systems (OSes). As a result, cross-platform implementation and deployment of acoustic sensing apps require a significant amount of time/effort.

We introduce below the three major challenges in building such cross-platform support.

Fragmented programming language support Porting the same signal processing algorithms and applications to different programming languages and platforms, such as Java

on Android and Objective-C/SWIFT on iOS, are difficult, time-consuming, and error-prone [51]. Moreover, any modification of the algorithms causes changes of code on multiple platforms, entailing significant development and maintenance costs.

Platform-dependent settings Each platform has a specific set of settings, so developers must acquire detailed knowledge to set the correct values on multiple platforms. For example, adding a recorder flag in Android (`AudioSource.VOICE_RECOGNITION`) might significantly change the acoustic sensing behavior by disabling the automatic gain control (AGC). A similar tweak exists in iOS's `AVAudioSession`. These settings can be easily overlooked by the developers unfamiliar with platform SDK.

Device hardware tuning Since real-time acoustic sensing has stringent timing and signal requirements, it is essential to tune the algorithms to different device hardware. For example, some microphones might even receive 20dB less signal strength at 22kHz than others [81]. It is also worth noting that, due to the varying installation location of the microphone/speaker on a device, a fixed-volume sound might saturate the microphone on certain devices while it might be too weak to be picked up on other devices.

We propose `LibAcousticSensing` (`LibAS`)[1] to meet these challenges by facilitating the rapid development/deployment of acoustic sensing apps on different platforms. `LibAS` is designed based on our study of more than 20 existing acoustic sensing apps. Specifically, large amounts of time and effort have been spent repeatedly to address the *common* platform issues that are often irrelevant to each app's sensing purpose. For example, several researchers [82, 100, 118, 132] mentioned the problem caused by non-real-time OS delay and then solved this issue by syncing the played/recorded sound through their own ad-hoc solutions. This repeated effort can be avoided or significantly reduced by providing a proper abstraction that handles the common platform issues systematically.

Fig. 5.1 shows the concept of `LibAS` which divides a normal acoustic sensing app into

[1] `LibAS` is open-sourced at: https://github.com/yctung/LibAcousticSensing

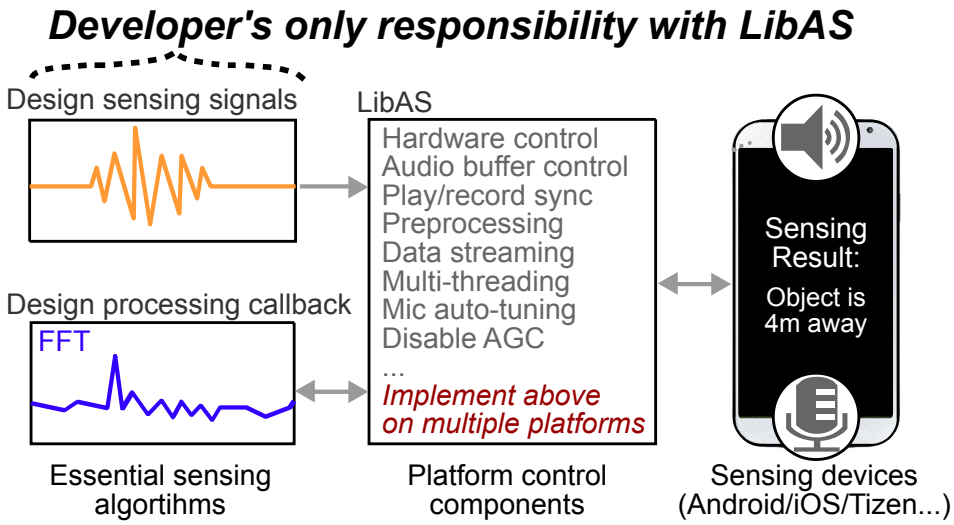


Figure 5.1: Concept of LibAS. `LibAS` reduces the cross-platform development effort for acoustic sensing apps by hiding laborious platform-dependent programming details.

two parts: (1) *essential sensing algorithms* and (2) *platform control* components. This separation is based on an observation that the design of sent signals (e.g., tones, chirps, or phase-modulated sounds) and the analysis of received signals (e.g., Doppler detection, fingerprinting, or demodulation) are usually aligned with the specific app's goal. Besides these two app-dependent components, handling audio ring buffers and audio multiplexing are mostly duplicated across apps and are closely related to the specific platform. Given such characteristics of acoustic sensing apps, separating the essential sensing algorithms from the other components can provide a cross-platform abstraction that hides the platform-dependent details from the app developers. To date, `LibAS`'s platform control API has been implemented to support iOS, Android, Tizen, and Linux/Windows.

With `LibAS`, app developers are only required to choose signals to sense and then build a callback function for handling each repetition of the sensing signals being received. The developers can either build the callback function by using `LibAS`'s server-client remote mode with Matlab, or choose the standalone mode with C (both are cross-platform supported). The former provides rapid prototyping environments (the received signals can be monitored and visualized via the built-in utility functions and GUI), while the latter provides a fast computation interface and can be shipped in a standalone app. We expect

developers to use the remote mode first to design and validate their sensing algorithms, and then transform their algorithms to the standalone mode, when necessary. Building acoustic sensing apps in this remote-to-standalone way not only reduces the development effort (compared to programming directly in the platform languages without any visualization support) but also makes the essential components *platform-agnostic*. This is akin to several well-known cross-platform UI libraries [11, 25, 27], but we are the first to apply it to acoustic sensing apps.

Evaluation results show that our library significantly reduces the cross-platform development effort of acoustic sensing apps. Specifically, `LibAS` has been used to build three demonstrative acoustic sensing apps. These apps cover the three major categories of acoustic sensing, i.e., sonar-like sensing, inter-device interaction, and sound fingerprinting. Our implementations show `LibAS`'s adaptability to meet all the unique requirements of these categories, such as real-time response, capability of controlling multiple devices, and connecting to a third-party machine learning library. With `LibAS`, app implementations require only about 100 lines of code to build (excluding the code for user interface). `LibAS` reduces up to 90% of the lines of code in the projects for which we acquired the source code. Three developers who used `LibAS` to build projects reported significant reductions of their development time, especially in case of building the first prototype, e.g., from weeks to a few days. As reported by these developers, `LibAS`'s utility functions ease the cross-platform/device tuning by visualizing several critical sensing metrics in real time. Only minimal overheads are imposed by `LibAS`, e.g., 30ms and 5ms latencies in the remote mode and standalone mode, respectively.

This paper makes the following four contributions:

- Design of the first library to ease the cross-platform development of acoustic sensing apps [Section 5.3];
- Implementation of three existing apps from different categories using `LibAS` [Section 5.4 / Section 5.5];
- Evaluation of the overhead of `LibAS` and its capability to handle the effects of het-

Sonar-like sensing		
System	**Purpose**	**Testbed**
FingerIO [94]	Gesture sensing	Galaxy S4 & wearables
SoundWave [58]	Gesture sensing	MAC Air & 3 laptops
AudioGest [110]	Gesture sensing	Galaxy S4/Tab & Mac
LLAP [127]	Gesture sensing	Galaxy S5
vTrack [47]	Drawing & Keystroke sensing	Galaxy Note4 & S2
ApneaApp [93]	Sleep apnea detection	4 Android devices
BumpAlert [120]	Obstacle detection	6 Android devices
Inter-device interactions		
PseudoRanging [81]	Location sensing	Special mic and amp
Spartacus [115]	Movement sensing	Galaxy Nexus
AAMouse [130]	Movement sensing	Google Nexus 4 & Dell laptop
SwordFight [132]	Device distance sensing	Nexus One & Focus
BeepBeep [100]	Device distance sensing	HP iPAQ & Dopod 838
DopEnc [131]	Life logging	6 Android devices
Dhwani [92]	Device communication	Galaxy S2 & HP mini
Sound fingerprinting		
EchoTag [118]	Location sensing	4 Android & 1 iPhone
RoomSense [109]	Location sensing	Galaxy S2
CondioSense [83]	Location sensing	4 Android devices
Acoustruments [79]	User interface augmenting	iPhone 5c
Symbolic Location [76]	Contacted surface sensing	Nokia 5500 Sport
Touch & Activate [97]	Touch sensing	Arduino
SweepSense [80]	Configuration sensing	Mac & earbuds

Table 5.1: Acoustic sensing apps. Most ubiquitous acoustic sensing apps are only implemented and tested on few devices and platforms. We categorize these apps into three types and will demonstrate how to build sensing apps of each type with `LibAS`.

erogeneous hardware/platform support [Section 5.6]; and

- User study of three developers who used `LibAS` in their real-world projects [Section 5.7].

5.2 Related Work

Acoustic sensing is expected to become ubiquitous as it can be integrated into existing computational platforms, such as laptops, mobile phones, wearables, and IoT devices. Sensing via acoustic signals can be either *passive* or *active*. Passive acoustic sensing only utilizes the device microphone to record the environmental sounds, user speeches, or the

sound of nearby events to provide context-aware computations [37,62,63,88,109,116,124]. On the other hand, active acoustic sensing uses both the speaker and the microphone to send a specially-designed sound and then monitor/evaluate how this sound is received/modified by the targeting events. Active acoustic sensing can provide virtual touch interfaces [47, 79, 97, 130], determine the relative movements between devices [81, 100, 115, 131, 132], remember the tagged locations [76, 109, 118], or recognize the users' status or their interactions with devices [58, 93, 94, 119]. `LibAS` is designed mainly for active acoustic sensing but it can also be used for passive acoustic sensing. Existing active acoustic sensing apps are summarized in Table 5.1. Although these apps are usually claimed to be *ubiquitous*, most of them are actually implemented and tested only on one or two types of devices. This lack of cross-platform support hampers the deployment of acoustic sensing apps. `LibAS` is designed to solve this problem by providing a high-level abstraction that hides device/platform-dependent development details from the acoustic sensing algorithms.

`LibAS` also provides utility functions to help tune the performance of acoustic sensing on devices. Existing studies reported several tuning issues caused by high speaker/microphone gain variations [81], lack of device microphone controls [124], speaker ringing effects [92], significant physical separation of microphones from speakers [119], random speaker jitters [132], and adaptation of microphone automatic gain control (AGC) [118]. These issues have usually been addressed in an ad-hoc way by each developer, and there are few systematic ways to analyze them. `LibAS` provides an easily accessible GUI to select microphones/speakers available to sense, determine if the AGC can be disabled, and then determine the gains of signals received between different pairs of microphones and speakers. We also plan to add a crowdsourcing feature that helps collect these types of device information when developers are testing them with `LibAS`, so that future developers will have a better insight before actually building their own acoustic apps.

There already exist several libraries/frameworks targeting the acoustic sensing apps, but all of them have very different goals from `LibAS`. For example, Dsp.Ear [54] is a

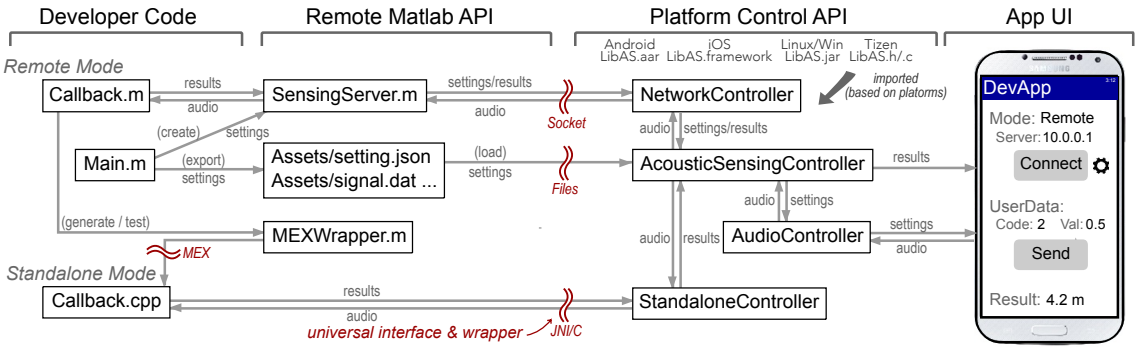


Figure 5.2: System overview. `LibAS` provides a universal interface/wrapper to communicate with the callback components. Thus, the platform control API can be easily imported to support different devices/platforms while keeping the developer's essential sensing algorithm consistent.

system-wide support to accelerate the acoustic sensing computation by utilizing phone's GPU while DeepEar [78] is a library for constructing a convolution neural network (CNN) based on acoustic sensing signals. Auditeur [96] and SoundSense [88] focus on providing a scalable/cloud-based service to collect acoustic features. CAreDroid [50] is an Android-only framework to provide an efficient sensing (including acoustic signals) interface for context-aware computations. The closest to `LibAS` is Code In The Air (CITA) [71, 107], which also provides a cross-platform abstraction for smartphone sensors (including microphones). However, CITA focuses on only *tasking apps*, such as sending a message to users' wives when they leave office, and has limited real-time support for active acoustic sensing. Note that most of these libraries/frameworks are also parallel to `LibAS`, so can be integrated in `LibAS`, if necessary. In terms of the cross-platform development, `LibAS` is more closely related to the well-known PhoneGap [25], ReactNative [27], and Cocoa2d-x [11] frameworks where developers can build cross-platform mobile app/game UI by JavaScript or C++.

5.3 System Design

Fig. 5.2 provides an overview of `LibAS`. As shown in this figure, among the four components of an app developed by `LibAS`, only the leftmost component includes the developer's

code that realizes the essential sensing algorithm of apps. The platform control API is the only platform-dependent component which needs to be imported based on the target platforms (e.g., LibAS.aar for Android or LibAS.framework for iOS). The main interface of `LibAS` exposed to developers is the class called `AcousticSensingController` which can be initialized to either a "remote mode" or a "standalone mode". In what follows, we will describe how to use `LibAS` in these two modes, how `LibAS` can be cross-platform supported, and the development flow of using `LibAS`.

5.3.1 Remote Mode

In the remote mode, the phone/watch becomes a slave sensing device controlled by a remote Matlab script called `Main.m`. This Matlab script creates a `LibAS` sensing server which will send the sensing sounds to devices, sync the developer's sensing configurations, and use the assigned callback function, `Callback.m`, to handle the recorded signals. The recorded signals will first be pre-processed by `LibAS`, truncated into small pieces (i.e., the segment of each sensing signal repetition), and each segment will be sent to the callback function. The callback function is responsible for calculating the sensing result based on the app's purpose. A conceptual callback function of a sonar-like sensing app can be:

$$dists = peak\_detect(matched\_filter(received\_signal))$$

Note the sensing results, e.g., `dists` in this example, will be automatically streamed back to the device for providing a phone/watch UI update (e.g., dumping the result as texts) in real time. A complete code example of using `LibAS` to implement real apps is provided in the next section.

Our remote mode design aims to help developers focus on building the essential sensing algorithm in a comfortable programming environment, i.e., Matlab. Note that our current Matlab implementation is only a design choice; it is possible to build the same function-

ality in other languages (e.g., Python). We choose to implement the remote mode with Matlab because it provides several useful built-in signal processing and visualization tools. Many existing acoustic sensing projects also use Matlab to process/validate their acoustic algorithms [37, 37, 76, 82, 89, 93, 115, 118–120, 130].

5.3.2 Standalone Mode

In contrast to the remote mode, the standalone mode allows sensing apps to be executed without connecting to a remote server. To achieve this, the developers are required to export their sensing configurations in the `Main.m` to binary/json files (with a single `LibAS` function call) and then transform the `Callback.m` function to C. In our current setting, this Matlab-to-C transformation can be either done manually by the developers or automatically completed by the Matlab Coder API [22, 31]. The transformation should be straightforward since `LibAS`'s abstraction lets the callback only focus on *"how to output a value based on each repetition of the received sensing signals"*. The C-based callback has the same function signature as our Matlab remote callback so it can be easily connected to the same `LibAS` platform control API. Specifically, when developers already have a workable sensing app with the remote mode, the standalone can be enabled by passing the C-based callback function as an initialization parameter to `AcousticSensingController`. The app will then seamlessly become a standalone app while all other functions work the same as in the remote mode.

Note that the standalone mode not only can execute the app without network access but also usually provide a better performance, such as a shorter response latency. However, it is challenging to develop/debug the sensing callback directly in the low-level language like C due to the lack of proper signal processing and visualization support. For example, if the developers incorrectly implement the previously-mentioned matched filter with wrong parameters, the detections might seem correct (still able to detect something) while the detec-

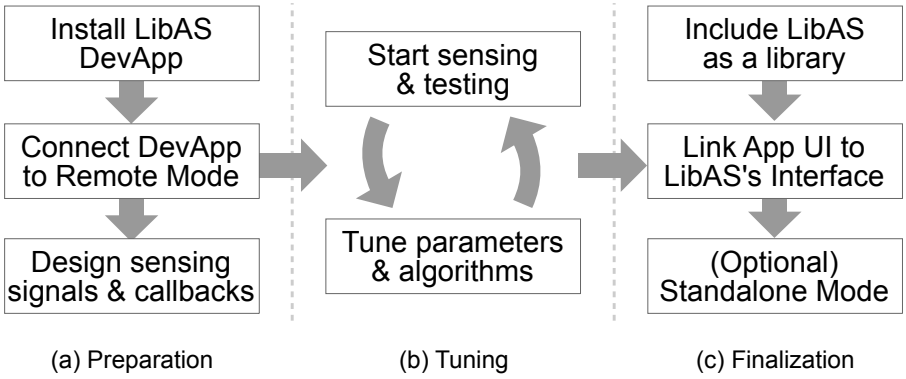


Figure 5.3: Expected development flow. Developers can first use the published `LibAS` DevApp (cross-platform supported) to realize their idea without even installing platform development kits, like XCode or AndroidStudio.

tion performance is severely degraded. To solve this problem, we provide a `MexWrapper`[2] that can easily connect/test the C callback even in the remote mode (i.e., the recorded audio will be streamed remotely to the Matlab server but the received signal will be processed by the C-based callback, as shown in Fig. 5.2). This "hybrid" mode helps visualize and debug the C callback. For example, the matched filter with wrong parameters can be easily identified by plotting the detected peak profiles in Matlab.

5.3.3 Expected Development Flow

Fig. 5.3 shows a typical development flow of using `LibAS`. New developers may first install our remote Matlab package and the pre-built DevApp (like an example developing app). This DevApp helps developers connect phones/watches to their sensing servers, with a simple UI as shown in Fig. 5.2, thus eliminating the need to install and learn platform development kits at this stage. After deciding which signals to use and how to process them in the callback function, the developers can start testing their sensing algorithms on real devices from different platforms and then modify the algorithm to meet their goals.

Developers can, in the end, choose to install a platform development kit for building the UI that they like to have and import `LibAS` as a library for their own apps. Note that the

[2]MEX [8] is an interface to call native C functions in Matlab

same sensing algorithm implemented in Matlab can be used for both ways, i.e., executing DevApp directly or including `LibAS` as an external library. Most of our current developers choose our remote mode to build their demo apps thanks to the simplicity and the strong visualization support in Matlab.

Once developers have validated their algorithms in the remote mode, they can (optionally) finalize the app to the standalone mode. As described earlier, developers can finish this transformation easily with the Matlab Coder API [22, 31], test with our Matlab `MEXWrapper`, and then ask the app to connect to the standalone callback function. Whenever developers notice a problem with their designed sensing algorithms, they can easily switch back to the remote Matlab mode for ease of the subsequent development.

5.3.4 Cross-platform Support

Some astute readers might have already noticed that `LibAS` can provide the cross-platform support because our platform control API connects to developers' sensing algorithms via several universal interfaces/wrappers (marked by the double tilde symbols in Fig. 5.2). For example, in the remote mode, different devices/platforms talk to the remote server via a standard socket. In the standalone mode, devices can understand the C-based callback either directly (like iOS and Tizen) or through native interfaces, like JNI/NDK (Android and Java). In summary, nearly all mobile platforms can understand/interface C and standard sockets, thus enabling `LibAS` to support the development of cross-platform acoustic sensing apps.

We are not the first to support cross-platform frameworks with this concept. For example, a well-known game library, Cocos2d-x [11], also helps build cross-platform games in C. Our remote mode model is also similar to several popular cross-platform UI frameworks like PhoneGap [25] and ReactNative [27] that load app UI/content from a remote/local JavaScript server. However, we are the first to apply this concept to acoustic sensing apps. With its cross-platform support, `LibAS` enables acoustic sensing apps to be ported to

various existing and emerging devices.

5.4 Implementation

`LibAS` implements the platform control API in Java, Objective-C, and C separately, and exports them as external libraries on different platforms, such as .aar for Android, .framework for iOS, .jar for Linux/Windows, and .h/.c for Tizen. These implementations follow a unified protocol to play/record acoustic signals and talk to the remote Matlab sensing server. Implementing this framework to offload the sensed data to a Matlab, process it, and then return the result to the device is challenging for the following two reasons.

First, it requires the domain knowledge of controlling audio and network interfaces on different platforms. For example, Android SDK can support recording and playing audio signals simultaneously by just writing or reading bytes from `AudioRecord` and to `AudioTrack` classes in separate threads. However, iOS requires developers to know how to allocate and handle low-level audio ring buffers in the `CoreAudio` framework to record audio in real time (i.e., fetching the audio buffer for real-time processing whenever it is available rather than getting the entire buffer only when the recording ends). Tizen needs its native programming interface of `audio-in/out` classes to record and play audios based on a callback. Also, different platforms usually need their own ways to control which microphone/speaker to use and how to send/record the signal properly. These development overheads can be the roadblock to the realization of existing acoustic sensing apps on different platforms.

Second, building a Matlab server to control the sensing devices via sockets is not trivial. Even though Matlab has already been used to process and visualize acoustic signals in many projects [37, 76, 115, 118, 119, 130], the lack of well-established socket support makes it challenging to realize our remote mode, especially when multiple devices are connected. For example, during the development of `LibAS`, we discovered and reported two issues with Matlab's socket library regarding randomly dropped packets and UI thread blocking.

Similar issues are also noticed by our current users. As a result, they chose to either export the audio signals as files or processing signals directly in Java before using LibAS. To address these issues that might have been caused by Matlab's single-thread nature, LibAS builds its own external Java socket interface. This Java socket interface is then imported to Matlab to support reading/writing multiple sockets simultaneously in separate threads. The performance of our current design is shown in Section 5.6.

5.5 Demonstrative Applications

We have implemented three different types of acoustic sensing apps with LibAS. These apps are chosen to cover the three major acoustic sensing categories: (1) sonar-like sensing (2) inter-device interaction, and (3) sound fingerprinting. Table 5.1 shows how existing projects belong to these three categories. The purpose of these implementations is to show how LibAS can be used to build real acoustic sensing apps and how LibAS can reduce the development efforts.

The first demo app is a sonar sensing on phones, which sends high-frequency chirps and estimates the distance to nearby objects based on the delay of received/reflected chirps. Technically, the development pattern of such a *sonar-like sensing* is akin to many existing approaches which send a sound and analyze the reflected echoes in real time (even though the signals might be processed differently). The second demo app is an inter-device movement sensing based on the Doppler effect [108]. This app can be regarded as a generalization of multiple existing *inter-device interacting* apps. We will use this app to show the advanced features of LibAS, such as how to control multiple devices simultaneously. The last demo app is a graphic user interface (GUI) which can easily classify different targeted activities based on acoustic signatures. The functionality of this app can cover several existing projects to know the location/status of phones by comparing the acoustic signatures. We will utilize this app to demonstrate Matlab's GUI support and capabilities in connecting to other 3rd-party libraries to quickly validate acoustic fingerprinting apps.

```
SERVER_PORT = 50005;
JavaSensingServer.closeAll();

% 0. sensing configurations
FS = 48000; PERIOD = 2400; CHIRP_LEN = 1200;
FREQ_MIN = 18000; FREQ_MAX = 24000;
FADING_RATIO = 0.5; REPEAT_CNT = 36000;

% 1. build sensing signals
time = (0:CHIRP_LEN-1)./FS;
signal = chirp(time, FREQ_MIN, time(end), FREQ_MAX);
signal = ApplyFadingInStartAndEndOfSignal(signal, FADING_RATIO); % for
    inaudibility
as = AudioSource('demo', signal, FS, REPEAT_CNT);

% 2. parse settings for the callback
global PS; PS = struct();
PS.FS = FS; PS.SOUND_SPEED = 340; PS.thres = 0.5;
PS.matchedFilter = signal(CHIRP_LEN:-1:1);

% 3. create sensing server with callback
ss = SensingServer(SERVER_PORT, @SonarCallback, SensingServer.
    DEVICE_AUDIO_MODE_PLAY_AND_RECORD, as);
```

Code 5.1: SonarMain.m. The remote main script defines the sensing settings and creates a sensing server.

5.5.1 Demo App: Sonar Sensing

The first app we developed with `LibAS` is a basic sonar system that continuously sends inaudible chirps to estimate the distance to nearby objects. The distance can be measured by calculating the delay of reflected echoes with the corresponding matched filter [112] (for the linear chirp case, the optimal matched filter is the reverse of sent chirps). We chose this app to illustrate the basic steps of using `LibAS` for the simplicity of the necessary signal processing. It can be easily extended to many other existing projects which also sense the environments/objects based on the real-time processing of the reflected sounds.

Code 5.1 (main script) and Code 5.2 (callback function) show how this sonar app is implemented in our remote Matlab mode. As mentioned earlier, the main script and callback are the only two required functions which developers need to implement for their sensing algorithms. As the main script shown in Code 5.1, we first create the desired chirp signals ranging from 18kHz to 24kHz and then pass this created chirp signal along with the desired `SonarCallback` function to the `SensingServer` class. A few other constants necessary for the callback to parse the received signal — e.g., the signal to correlate as the matched

```
function [ret] = SonarCallback(action, data, user, context)
  global PS; % parse settings
  USER_CODE_RANGE = 1;

  % 1. init callback parameters
  if action == context.CALLBACK_INIT
    PS.detectRange = 5; % meter

  % 2. process the senisng data
  elseif action == context.CALLBACK_DATA
    cons = conv(data, PS.matchedFilter);
    peaks = cons(cons > PS.thres);
    dists = 0.5 * (peaks(2:end) - peaks(1)) * PS.SOUND_SPEED / PS.FS;
    dists = dists(dists < PS.detectRange);
    ret = SenisngResult(dists, 'floatArray');

  % 3. user-specificied events (optional)
  elseif action == context.CALLBACK_USER && user.code ==
     USER_CODE_RANGE,
    PS.detectRange = user.intVal;
  end
end
```

Code 5.2: SonarCallback.m. The remote callback focuses on processing each repetition of the sensing signals received.

filter in this example — can be assigned to a global variable called `PS`.

As the remote callback function shown in Code 5.2, the received `data` argument can belong to 3 different types of `action`. When the server is created, an action called `CALLBACK_INIT` will be taken to initialize necessary constants/variables. In this example, we assign the value of `detectRange` and it can later be updated in the app's UI. The most important part of the callback function occurs when the received data belongs to the action `CALLBACK_DATA`. In this case, the received data will be a *synchronized* audio clip which has the same size and offset as the sent signal. For example, it will be the 25ms chirp sent by the app plus many reflections following. The synchronization (e.g., knowing where the sent signals in the received audio start) is processed by `LibAS` and hidden from the developers. So, developers can just focus on *how to process each repetition of sent signals received*. This is found to be a general behavior of acoustic sensing apps [93, 94, 110, 118–120], where the apps usually focus on the processing of each repetition of the sent signal. Some system may need the reference of previously-received signals which can be done by buffering or using the `context` input argument. Details of these advanced functions can be found in `LibAS`'s Github repository [17].

Since the recorded signal has already been synchronized and segmented by `LibAS`, the processing of each recorded chirp is straightforward. As shown in Code 5.2, a matched filter, i.e., `conv()`, can be directly applied to the received signal, and then the peaks (i.e., the indication of echoes) are identified by passing a predefined threshold. The distance to nearby objects can be estimated by multiplying half of the peak delays (after removing the convolution offset) to the sound of speed, and then divide by the sample rate. The objects within the detect range will be added to a return object and then be sent back to the device for updating the app UI. Moreover, the simplicity of callback in `LibAS` makes its transformation straightforward for the C-based standalone mode (it can even be automated with the Matlab Coder API [22, 31]). Note that we intentionally make the detect range modifiable in this example to show the extensibility of `LibAS`. In this example, this value can be adjusted in the app by calling the `sendUserEvent(code, val)` function in our platform control API. An example of sending this user-defined data can be found in the DevApp UI as shown in Fig. 5.2. This extensibility is important for developers to customize their own sensing behavior with `LibAS`. For example, one of our current developers uses this function to send movement data (based on accelerometers) and then improve the acoustic sensing performance by integrating the updated movement data. More details of user experience of `LibAS` can be found in Section 5.7.

5.5.2 Demo App: Inter-Device Movement Sensing

The second demo app implemented with `LibAS` is the inter-device movement sensing app based on the Doppler effect. Specifically, the frequency shift due to the Doppler effect can be used to estimate the velocity and distance of movements between devices. Sensing the movement by Doppler shifts has been used to provide a virtual input interface to IoT devices [130], create a new gesture control between smartphones [115], and detect if two users are walking toward each other [131]. Our demo app implemented with `LibAS` can be viewed as a generalization of these movement sensing apps.

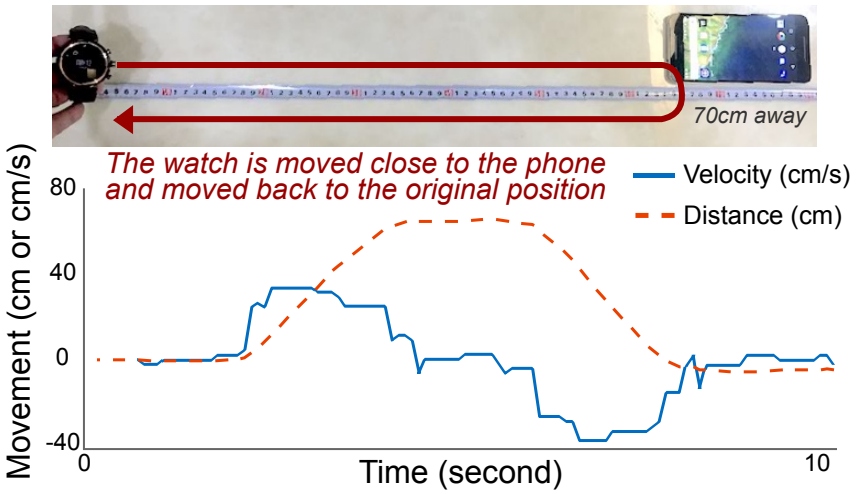


Figure 5.4: Movement sensing by Doppler shifts. The integrated area of velocity indicates the movement shift. A demo video can be found at `https://goo.gl/AiJba9` [19]

Based on Doppler's theory [108], the relationship between the changed movement speed and the shifted central frequency can be expressed as: $f_c' - f_c = v f_c / c$, where the f_c is the central frequency of sent signals, f_c' is the shifted central frequency, v is the relative velocity between devices, and c is the speed of sound. Note this Doppler detection is usually coupled with a downsampling and overlapped sampling to further improve the sensing resolution and response latency [115, 130, 131]. For example, we set the downsampling factor to 8 with an 87% overlapping ratio. This setting can provide a movement sensing resolution of 2cm/s instead of 17cm/s when sensing via a 20kHz tone.

Most of our current implementation of this demo app follows a similar pattern as in our previous demo app. Some minor changes include using different signals (i.e., narrow-band tones) and passing different parsing parameters to the callback (i.e., downsampling factors). The largest difference is to initialize multiple `SensingServer` classes for controlling multiple devices to sense simultaneously. For example, in this example, we have a transmitting/receiving server that connects two devices where one is responsible for sending the tone sound while the other is responsible for receiving and processing it. In `LibAS`, developers can easily configure multiple devices getting connected and then trigger the sensing among devices together. We omit the description of how to process the callback function of

the receiving device since it is nearly identical to the previous demo app except for applying FFT to the `data` argument instead of the matched filter.

Fig. 5.4 shows the result of moving a Zenwatch 3 from 70cm away toward a Nexus 6P and then moving it back to the original location. In this example, integrating the estimated velocity can estimate the moving distance as 64cm, which is about 6cm different from the ground truth. Similarly to the previous demo app, this implementation needs only about 100 lines of code to write and it can be easily executed on devices from different platforms, thus showing the simplicity of creating acoustic sensing between devices with `LibAS`. Note that this example can also be extended to other apps that need inter-device sensing, such as aerial acoustic communication or localization [81, 82, 92, 125]. A video of this demo app can be found at `https://goo.gl/AiJba9` [19].

5.5.3 Demo App: GUI for Activity Fingerprinting

The last demo app we have implemented is a graphic user interface (GUI) that can classify various activities based on acoustic signatures. This demo app is based on the property that different activities, like placing the phone at different locations, will result in different frequency-selective fading because the reflected acoustic signals are different from one location to another. This acoustic signature has been utilized in many sensing apps. For example, it can be used to distinguish rooms [109], remember the tagged locations [118], recognize the touched surface [76], and sense how the phone is touched [97]. Using the GUI of this demo app implemented with `LibAS`, similar fingerprinting apps can be realized without even writing a single line of code.

The GUI of this demo app built upon `LibAS` is shown in Fig. 5.5. After the device's DevApp connected to the Matlab server, developers can click the *new* button to create the target activity to sense. For example, we have used this GUI to implement a novel app of sensing hand gestures (i.e., a palm or a fist) with smartwatches. After these targets are created using the GUI, we ask users to place the hand on each target gesture and click

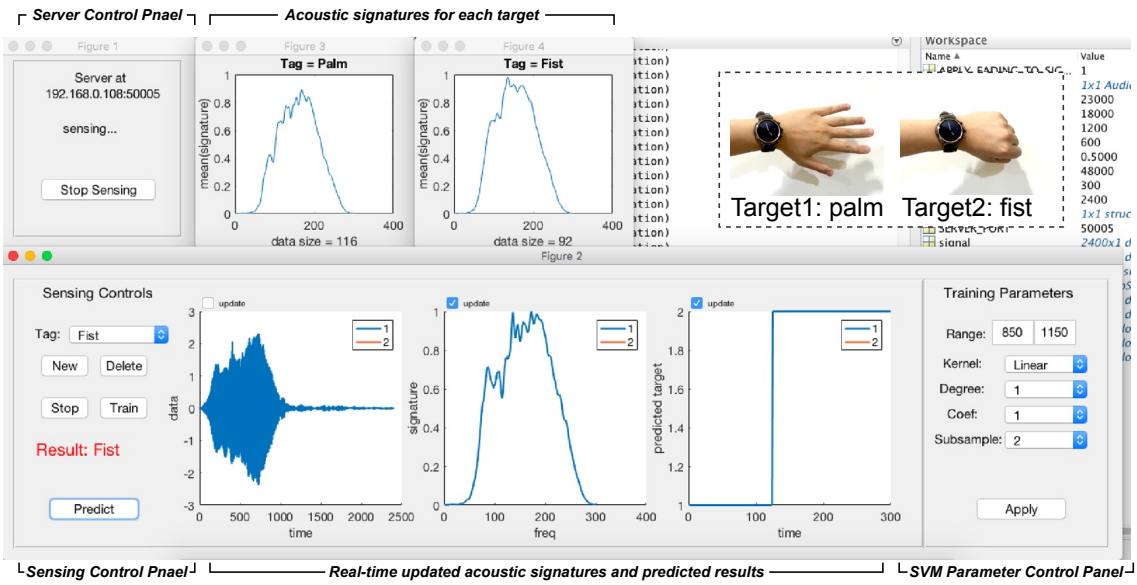


Figure 5.5: Graphical User Interface (GUI) for fingerprinting acoustic signatures. Developers can easily classify different user-defined actions based on acoustic signatures. A demo video of this GUI support can be found at `https://goo.gl/DqFFcA` [18].

the *record* button, which triggers the callback function to save the audio fingerprint of that target activity. The collected fingerprint is shown automatically in the GUI when the recorded data is streamed by `LibAS`, so that developers may easily see if it is possible to get reliable acoustic signatures for each target activity. SVM classifiers can be built by clicking the *train* button and the corresponding training parameters can be adjusted in the right side of the control panel. Once the *predict* button is clicked, the result of classification will be updated as the red text shown in the GUI in real time. This simple GUI for fingerprinting apps is shown to be able to identify the above-mentioned gestures by Zenwatch 3 with a higher than 98% accuracy. A demo video of this GUI can be found at `https://goo.gl/DqFFcA` [18].

This demo app shows how the GUI-support of Matlab and the capability of integrating 3rd-party libraries, i.e., LibSVM [43], can help develop acoustic sensing apps using `LibAS`. We have also used this GUI to realize other fingerprinting apps, such as EchoTag [118], and obtained reasonably good results in a very short time without any coding. This GUI can be easily adapted to passive acoustic fingerprinting apps, such as Batphone [116] and

others [37, 88], which use the sound of environments as acoustic signatures, rather than the sensing signals sent by the device.

5.6 Evaluation

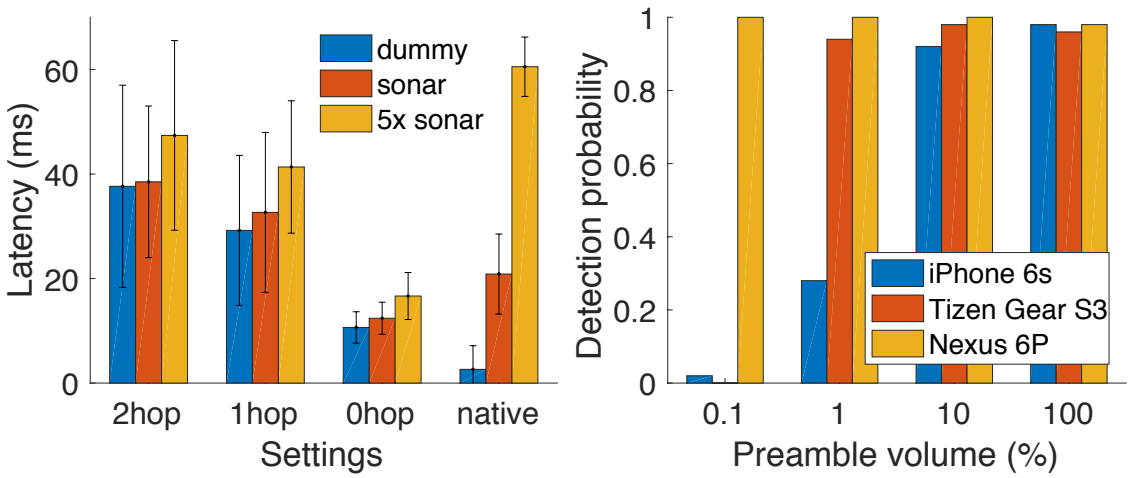
We will first evaluate the overhead of our current LibAS implementation, such as the response latency and the preprocessing cost. Then, we will show the adaptability of using LibAS to build cross-platform acoustic sensing apps. Specifically, we will show how the platform/device heterogeneity can affect our demo apps and how these issues can be identified/mitigated by using LibAS.

5.6.1 Overhead

As the system described earlier, LibAS is a thin layer between the developers' sensing algorithms and the sensing devices. In the standalone mode, the performance overhead of LibAS is nearly negligible since every component is built natively. In the remote mode, there can be an additional overhead caused by the network connection.

5.6.1.1 Response latency

LibAS provides a convenient utility function to measure the response latency of the developers' sensing configurations. We define the *response latency* of LibAS as the time that sensing devices record a complete repetition of sensing signals to the time that the sensing results based on this repetition are processed and returned from callbacks. Fig. 5.6(a) shows this latency profiles under different configurations. We first show a dummy case where the callback doesn't process anything, but returns a constant (dummy) sensing result instantly. This dummy case is used to characterize the baseline overhead that is caused by LibAS's implementation. As shown in the figure, we repeated the latency measurements under four different configurations, which include (i) 2-hop (the sensing server and the device are connected to the same 802.11n WiFi router), (ii) 1-hop (the sensing device is connected



(a) Response latency (b) Preamble detection

Figure 5.6: Overheads. The minimal overhead incurred by `LibAS` can support most real-time acoustic sensing apps.

directly to the server), (iii) 0-hop (the sensing device is connected to a server residing in the same machine), and (iv) standalone (the processing is executed natively and locally). In this experiment, we always use a 2013 MacBook Air, i.e., 1.3 GHz CPU & 8GB RAM, as the sensing server and an Android Nexus 6P as the sensing device (except the 0-hop case where the sensing device is the MacBook itself). The results from our implementations on iOS and Tizen follow a similar pattern but are thus omitted due to the space limit.

As shown in Fig. 5.6(a), without considering the processing cost of the sensing algorithm (i.e., a dummy case), the remote mode of `LibAS` can achieve an average response latency less than 40ms in a common 2-hop setting. This latency can be reduced further to 30ms by turning either the laptop or phone into a hotspot and connect them directly. Note that this 1-hop (hotspot) setting is particularly helpful to use `LibAS` when WiFi is not available, e.g., testing our sonar demo app in an outdoor environment. When both the sensing devices and the server are located on the same machine (i.e., zero networking overhead), the response latency can be reduced further to 10ms. By closely anatomizing the delay, 6ms comes from the way our Java socket interface is hooked to Matlab while only the other 4ms is caused by our pre-processing. According to our preliminarily test, implementing the same remote function directly on Matlab's asynchronous socket API incurs a >250ms

overhead under the same configuration, thus making it nearly impossible to provide a real-time response. Even though our standalone mode can push the response latency to less than 5ms, this 0-hop setting is useful for developers to build/test their sensing algorithms with the strong Matlab support while keeping the latency overhead at a level similar to the native calls. Generally, the low latency overhead of `LibAS` can meet the requirement of many real-time acoustic sensing apps, such as our sonar demo app that needs the processing to be completed in 50ms (i.e., before sending the next 2400-sample signals sampled in 48kHz).

It is important to note that, by considering the callback processing delay of the developer's sensing algorithms, the remote mode might sometimes get an even better overall performance than the standalone mode. For example, if we intentionally repeat the same sonar processing 5 times in the demo app callback, i.e., the 5x sonar case in Fig 5.6(a), the standalone mode will miss the 50ms deadline while the remote mode will not. This phenomenon is caused by the fact that a remote server (laptop) usually has more computation resource than the sensing device (phone/watch). This fact is widely used in offloading several computation-hungry processing, e.g., video decoding, to a cloud server [105] and might be necessary to build sophisticated acoustic sensing apps in future. `LibAS` already supports both modes and can automatically collect/report the performance overheads, so developers can easily choose whatever mode fits best to their purpose. As we will discuss in Section 5.7, the minimal latency overhead of `LibAS` meets the real-time requirements of our current users.

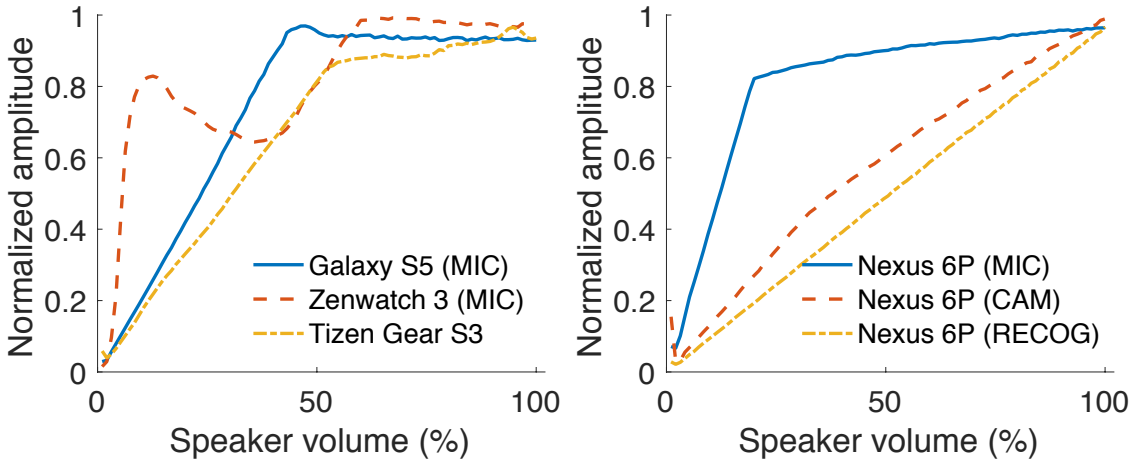
5.6.1.2 Preamble detection overhead

Preamble detection is an important feature provided by `LibAS` as it helps identify when the start of sent signals is recorded and then truncate the received audio signals into correct segments with the same size/offset of the sent signals. Our preamble detection is based on a similar design to existing approaches [82, 118, 119]. Specifically, the preamble is a series of chirps (must be different from the sensing signals) that can be efficiently identified by

a corresponding matched filter [112]. The performance of preamble detection depends on the length/bandwidth of preamble signals, the detection criteria, and also on the hardware capability. Theoretically, a long and wide-band preamble usually has a high detection rate, but also incurs a long initial sensing delay (i.e., the delay before playing the sensing signals) and more audible noise. We currently use 10 repetitions of a 15kHz–22kHz chirp plus a 4800 sample padding as the default preamble for `LibAS`. Each chirp is 500 samples long and separated from the next chirp by 500 samples. We set the criteria to pass the detection when all 10 chirps are detected correctly (i.e., jitter between detected peaks is less than 5). This ad-hoc setting is chosen based on our experience that can support most devices from different platforms for reliable detection of the start of signals.

We have conducted experiments of our current preamble detections on more than 20 devices. For the acoustic sensing apps that sense the signal sent by itself, such as our sonar demo app, `LibAS` can easily achieve higher than a 98% detection probability when 20% of volume is used to play the preamble. On the other hand, for apps that sense the signal sent from another device, like our inter-device demo app, `LibAS` can successfully detect a device 1m (5m) away with a higher than 90% (80%) probability. Our current design can reliably detect the preamble sent from a device 10m away when the retransmission of preamble is allowed. As shown in Fig 5.6(b), this performance might vary from device to device due to their hardware differences. This is the reason for our choice of a wideband 15kHz–22kHz preamble, which incurs fewer hardware artifacts and frequency-selective fading as shown in the following sections.

Note that this current preamble setting might not be perfectly aligned with every sensing app. For example, apps based on frequency-domain response rather than time-domain information, might be more tolerant of the segment errors. In such a case, developers might want to loosen the detection criteria to sense devices within a longer range or reduce the chirp bandwidth to make the preamble inaudible. The preamble parameters can be easily set by `AudioSource.setPreamble()` function, and it can be efficiently tested through `LibAS`.



(a) AGC on (b) Different flags are used

Figure 5.7: Automatic gain control detections (AGC). `LibAS` detects if AGC is enabled by sending a signal with linearly increased volumes.

We expect a better preamble design to emerge once more developers start building apps with `LibAS`. One of our current active developers using `LibAS` has modified our setting to a customized short and inaudible preamble since he only targets high-end smartphones with reliable microphone/speaker hardware while another developer added a longer padding period after the preamble to avoid overlapping it with the sensing signals when more than 3 devices are connected.

5.6.2 Adaptability

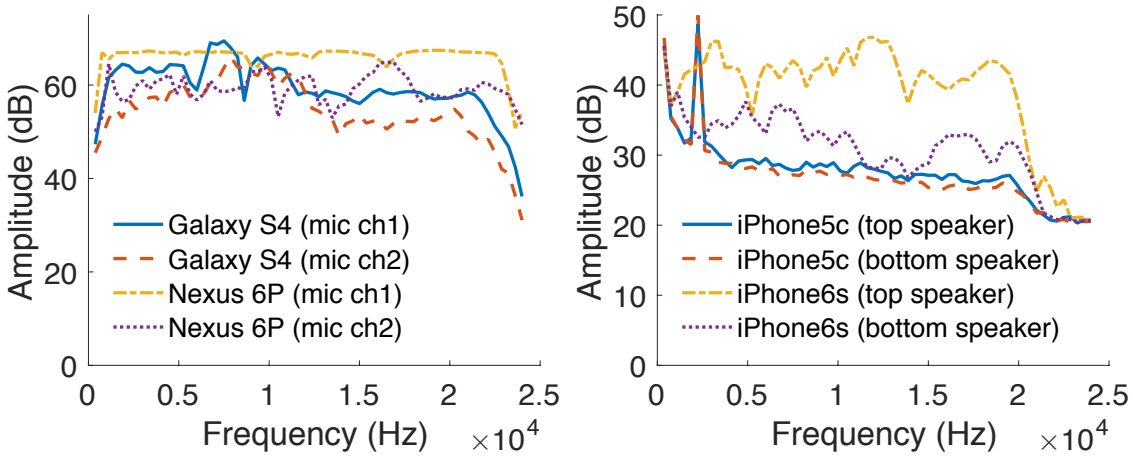
Adaptivity is an important performance indicator of `LibAS` since it is designed to support various applications, platforms, and devices. Our demo apps have shown that `LibAS`'s design is general enough to support several categories of acoustic sensing algorithms. Some other real-world use-experience of `LibAS` by our current developers will also be presented in the next subsection. Here we will focus on how `LibAS` can adapt the platform/device heterogeneity to improve the sensing performance and help developers make the correct sensing configurations.

5.6.2.1 Platform heterogeneity

One of the key software features of mobile OSes that might significantly affect the performance of acoustic sensing apps is the automatic gain control (AGC). AGC is reasonable for voice recording because it can avoid saturating the limited dynamic range of the installed microphone. However, AGC is usually not desirable for acoustic sensing apps. Taking our fingerprinting demo app as an example, AGC can alter the acoustic signatures when the ambient noise increases, thus reducing the prediction accuracy. AGC will also confuse the sonar demo app because the change of time-domain response might be dominated by AGC rather than the object reflections.

Fig. 5.7 shows `LibAS`'s utility function to identify AGC by sending a 2kHz tone over 4 seconds with a linearly increased speaker volume (from 0% to 100%). If the AGC is not enabled, the amplitude of received signals should increase linearly over time (in the same way as how the signal is sent). A few examples with AGC enabled can be found from Fig. 5.7(a), where the amplitude of received signals does not increase linearly and it stops increasing after the speaker volume reaches certain ranges.

In iOS, this AGC can be turned off by setting `kAudioSessionMode_Measurement`, but based on our experiments, the official Android AGC API always fails to disable the microphone AGC (e.g., not functional or indicating the API not implemented by the manufacturer). By using `LibAS` to loop several audio settings automatically, the response is found to vary based on the programming `AudioSource` flag set to the microphone, e.g., `MIC`, `CAMCORDER`, or `VOICE_RECOGNITION`. As shown in the example of Fig. 5.7(b), setting the flag to `VOICE_RECOGNITION` in Nexus 6P can disable AGC and make the response linear in the speaker volume. Based on our experiments, setting the flag to `VOICE_RECOGNITION` can actually turn off AGC for most Android devices we have tested except for Zenwatch 3. Our DevApp has a clear UI, helping developers to select these hidden platform options, check the effects of each option, and adapt their sensing configurations accordingly (e.g., using the sound volume in the linear range even when the AGC cannot be disabled).



(a) Different microphones (b) Different speakers

Figure 5.8: Frequency responses of various devices. The sensed frequency responses vary not only with devices but also with the microphone/speaker used to sense.

5.6.2.2 Device heterogeneity

The microphones and speakers in commodity phones are usually not designed for acoustic sensing, especially for inaudible sensing signals. Fig. 5.8 shows the frequency response of several Android and iOS devices. These results are based on experiments of sending and receiving 5 repetitions of a 4-second frequency sweep from 0Hz to 24kHz by the same device. This experiment was conducted in a typical room environment and the phone was held in the air by hands, which resembles the most common scenario of using smartphones. A similar experiment was done previously [81], but only the microphone gains were reported.

As shown in Fig. 5.8, sensing signals at certain frequencies could inherently have a 20dB higher signal strength than at other frequencies. For apps that need to detect frequency shifts due to Doppler effects, it would be more beneficial to sense in the range with flat frequency responses. Otherwise, acoustic sensing apps would prefer sensing at frequencies with stronger responses. Among the device we tested, Nexus 6P and iPhone 5c are the best to have consistent responses over different frequencies. We also noticed that iPhones generally have a lower speaker/microphone gain than Android devices, which could be due to the different hardware configuration in iOS. The strong peak of iPhone 5c on 2250Hz (as

shown in Fig. 5.8(b)) is usually known as the *microphone resonant frequency*. Most acoustic sensing apps should avoid sensing at this resonant frequency because the responses might be dominated by this effect, rather than by the target activity to sense.

This hardware heterogeneity calls for a careful design of sensing signals and algorithms. For example, according to our experimental results, the same sensing algorithm in our sonar demo app allows Nexus 6P, Galaxy S7, Galaxy Note4 to sense a glass door 3m way reliably (with more than 15dB SNR) based on a18kHz–22kHz inaudible sound. However, iPhone 6s, Galaxy S4, and Gear S3 are unable to achieve a similar performance unless using an audible sound. With `LibAS`, such device heterogeneity can be easily observed and adapted with our real-time visualization support.

5.7 User Experience

To evaluate the applicability of `LibAS`, we collected and analyzed feedback from three users (including experienced and novice developers) who were using `LibAS` for their projects. The first user (`EngineerA`) is an engineer at a major smartphone manufacturer, which sells more than 200M smartphones per year. He wanted to build a demo app for an existing sound-based force sensing project called ForcePhone [119]. The other two users are CS PhD students at different universities (`StudentB` and `StudentC`). They both wanted to build new acoustic sensing apps. `StudentB` is familiar with Android/Python, while `StudentC` has only relevant experience in processing WiFi signals with Matlab. `StudentB` had a proof-of-concept app before adapting this project to `LibAS`, and `StudentC` started her project from scratch with `LibAS`.

`EngineerA` was the first user of `LibAS` and collaborated with us. Since `LibAS` was not mature at that time, we provided him numerous technical supports and also added new functions based on his need. An issue that `EngineerA` faced was the hanging of `LibAS` installation, i.e., when adding our customized socket interface. This issue was later identified as a path problem and solved in our new update. `StudentB` is the second user of

LibAS. At the time StudentB used LibAS, we had already published DevApp online and documented our API. StudentB knew us but never worked on the same project. StudentB mostly worked independently to adapt his project to LibAS. StudentC is our third user and had no knowledge of our team or LibAS. She contacted us after seeing some of our publications on acoustic sensing apps. At the time StudentC started using LibAS, our installation guide and examples had already been documented and made available, so she can successfully install LibAS by herself and used our demo apps. Our study shows that LibAS not only significantly reduced the development efforts in real-world projects for experienced users but also lowers the bar for developing acoustic sensing applications for novice users.

5.7.1 Why LibAS?

Before using LibAS, EngineerA already had the Android source from the ForcePhone project team, but he found it challenging to modify the sensing behaviors directly in this source. For example, EngineerA had no idea why/how the program failed to work after his modifications on the sensing signals since all the processing was embedded in the native Java/C code. EngineerA wanted to build his system using LibAS because he needed full control of the ForcePhone project to meet his demo need.

StudentB's goal was to develop a smartwatch localization based on sounds (called *T-Watch*). Specifically, the app could estimate smartwatches' locations by triangulating the arrival time of signals recorded in paired smartphones. He already had a proof-of-concept system before using LibAS. Specifically, he developed an app on Android that played/recorded sounds simultaneously and saved the recorded sounds to a binary trace file. He loaded this file to a laptop via USB cable and then processed his sensing algorithm offline in Python. StudentB started using LibAS because he noticed that porting his proof-of-concept Python code to a real-time demo would be time-consuming and error-prone.

StudentC was developing an acoustic sensing app similar to our sonar demo app, but

based on an advanced FMCW sonar technique [75]. She had tried to build a standalone app on Android/iOS to process the FMCW directly but got stuck on several issues. For example, she was wondering why her own app could not record the high-frequency responses and why the played high-frequency chirp was audible. In fact, these were frequently asked by new acoustic sensing developers. She wanted to use `LibAS` because our demo apps could serve as a reference design to build her own project.

5.7.2 Benefits of Using `LibAS`

The biggest common benefit reported by all three users was the visualization provided by `LibAS`'s remote mode. They commented that such a visualization helped them "see" and understand how the received response changed (e.g., a higher/lower SNR) when the sensing settings were modified. Specifically, using this visualization, `EngineerA` tuned the system performance and investigated how the environmental noise affected his demo app; `StudentB` identified and solved several unseen issues such as the signal degradation caused by different wearing positions of smartwatches; and `StudentC` learned the sonar sensing capability on real devices and also noticed the potential issues caused by reflections from ceilings/floors.

`LibAS`'s abstraction was another benefit these three users mentioned multiple times. They all reported that this abstraction helped them focus on developing the core sensing algorithm, so they could build/transform their projects quickly. Note that our abstraction is designed with the consideration of future extensibility. For example, in `StudentB`'s project, the sensing algorithm needed to know if users were clicking a surface based on accelerometer data, and these "additional" data could be easily sent by the extensible user-defined data interface of `LibAS`, as described in Section 5.5.

The cross-platform support might not be the primary incentive for these three developers to use `LibAS`, but it turned out to be an unexpected handy feature the users enjoyed. For example, `StudentB` was targeting only Android phones and smartwatches, but he later

ForcePhone (no UI)		T-Watch (offline)		BumpAlert (no camera)	
Java	3913	Java	4080	Java	1734
Matlab	1992	Python	1998	Python	765
C/C++	2273	C/C++	115	C/C++	353
w/ LibAS		w/ LibAS		Sonar demo app	
Matlab	490	Matlab	1390	Matlab	279
Java	446				
C/C++	153				
Reduction	86%	Reduction	78%	Reduction	90%

Table 5.2: Estimated code reductions. The significant reduction of code demonstrates the capability of `LibAS` to save development time/effort.

discovered that one strong use-case for his project was to provide a touch interface on laptops by directly using the laptop microphones. If his app had been built natively on Android, it would be another challenging task to transform the app to Linux/Windows, but this was not a problem with `LibAS`. By using `LibAS`, he easily achieved this by installing DevApp (including the platform control API) on his laptop. He also used this function to test his project on a newly-purchased Tizen smartwatch. The same benefit was also seen by `StudentC`, when she installed DevApp on her personal iPhone and conducted a few experiments with it. We expect the cross-platform support to become a more attractive feature when users notice the effort of cross-platform development is significantly reduced by using `LibAS`.

5.7.3 Estimated Code Reduction

We analyze the code reduction of these three use-cases to estimate the potential saving of effort when using `LibAS`. Note that this estimation is not perfectly accurate since the apps developed with `LibAS` by our users are not completely identical to the original/reference implementations. For example, the ForcePhone source code includes some demo/test activities, but they are not necessary for `EngineerA`. There are also no real-time processing components in `StudentB`'s original offline Python code since he built the real-time demo app only with `LibAS`. We have tried our best to estimate the (approximate) code

reduction fairly by ensuring the code w/ or w/o LibAS implements the same functionality. Specifically, we remove all components from the original/reference system that are not used in our users' implementations with LibAS (such as the demo/test activities). We use our sonar demo app and an existing sensing app called BumpAlert [120] as the reference for StudentC's case since StudentC has not yet finished her project. The trend of code reduction should be similar if she keeps building her system upon our sonar demo app.

Table 5.2 shows the lines of code estimated with the open-sourced cloc tool [35], where all three use-cases are shown to have more than 78% reduction of code. Most of the reduction comes from the abstraction of hiding platform-dependent code. EngineerA is the only one who includes LibAS's platform control API in his own app while others use our prebuilt DevApp. Even in this case, his app's Java implementation is still significantly shorter than the original design (490 lines instead of 3913) since most audio/network/processing-related components are covered by LibAS. Note that this reduction should be even more prominent if we consider StudentB's and StudentC's use of LibAS on Linux/Win, Tizen, and iOS devices.

After taking a close look at StudentB's original implementation, we notice that 1745 lines of code are used to handle the audio recording and 370 lines of code are used for communicating the recorded trace. These lines of code are reduced to 62 and 22 lines with LibAS, respectively. The code for the UI is nearly identical for both implementations (w/ or w/o LibAS), but there is a large reduction of code on processing the received signals. He told us this reduction exists because he sometimes needed to duplicate his Python processing on Android to check some performance metrics before going to the offline parser. This problem does not exist when LibAS is used since it allows the designed algorithm to be executed on different platforms. Even though our estimation is not general, it is still an interesting indicator of how LibAS can save developers' efforts in building acoustic sensing apps on different platforms.

5.8 Discussion

LibAS is a cross-platform library to ease the development of mobile acoustic sensing apps. However, when developers want to release their production apps, they might still need to implement the customized UI on multiple platforms by using the specific platform development kits. One way to solve this problem is to package LibAS as a native plug-in for existing cross-platform UI libraries, such as PhoneGap [25] or ReactNative [27]. Some other features that we are considering to add, include the support of Apple's Watch OS4 and Tizen TV, more demo examples, and a crowdsourcing platform to share developers' sensing configurations. Also, there is potential for LibAS to support "non-sensing" apps, such as real-time sound masking or authentication based on acoustic signal processing [52, 72].

LibAS has been open-sourced for months. We expect the open-source community to try and refine the idea of LibAS. For example, based on EngineerA's suggestion, we added a save/replay function that allowed developers to keep their sensed signals in a file and then replay it with their assigned callback. This function is useful since developers might want to try different parameters/algorithms based on the same experiment data. StudentB has become an active contributor, and helped build several useful UI features on our DevApp. We expect to see more changes like these, so LibAS may help acoustic sensing apps truly ubiquitous.

5.9 Conclusion

We have presented LibAS, a cross-platform framework to ease the development of acoustic sensing apps. Developing acoustic sensing apps with LibAS is shown to reduce lines of code by up to 90% and provide cross-platform capabilities with minimal overheads. LibAS has been open-sourced and currently supports Android, iOS, Tizen, and Linux/Windows. Three developers have already used LibAS to build their own apps and they all agree that it saves their effort significantly in developing acoustic sensing apps.

CHAPTER VI

Conclusion and Future Works

In this thesis, we proposed and implemented three novel acoustic sensing applications, `BumpAlert`, `EchoTag`, and `ForcePhone`. These apps rely only on acoustic signals and built-in hardware, so they are compatible with most existing mobile devices and likely to remain that way in the future. From our experience of designing these three apps, we learned that building acoustic sensing apps usually requires a significant amount of specific domain knowledge and faces challenges unseen in traditional apps. Thus, to facilitate the development of acoustic sensing apps, we proposed a cross-platform framework called `LibAS`. `LibAS` provides a proper abstraction that separates the core sensing algorithms from platform hardware controls, thus allowing developers to prototype/test their design quickly without even installing the platform development kits. Since apps built by `LibAS` are cross-platform supported, they can be installed/executed on multiple popular mobile platforms, such as Android, iOS, or Tizen, without modifying their code. The simplicity/efficiency of developing acoustic sensing apps with `LibAS` can help non-experts with various design backgrounds to develop new acoustic sensing apps.

6.1 Acoustic Sensing Applications

Since audio is a natural interface for humans to interact with devices, microphones and speakers have become the standard/common built-in sensors on nearly all mobile/wear-

able/IoT devices. Acoustic sensing apps reuse these built-in microphones/speakers to provide advanced sensing functions. For example, `BumpAlert` has shown that acoustic sensing in mobile devices can detect objects more than 4 meters away, and `EchoTag` has shown that the sensing resolution can distinguish locations even at a cm-level. `ForcePhone` further utilizes the sounds traveling through the smartphone body to provide a force-sensing interface.

Even though the hardware of built-in microphones/speakers cannot achieve the same sensing capability as specially-designed sensors (like SONAR or LIDAR), their *hardware* limitations can usually be mitigated by the proper *software* design. For example, `BumpAlert` removes the unnecessary reflections from the floor and ceiling by matching the users' walking speeds to acoustic reflections. `EchoTag` utilizes the drawn outline of phones to compensate for a small placement tolerance. `ForcePhone` provides an animation clue to indicate the level of estimated force, so users can easily adjust their behavior even when the estimation drifts. The broad sensing range and fine sensing resolution shown in this thesis can also be used in many other sensing functions. Users might be able to communicate or interact with others using the acoustic signals, life-log or monitor the environments based on acoustic signatures, or even determine the holding posture or the user's identity based on the damped acoustic sensing signals. Several similar concepts have been implemented in recent years [79, 83, 110, 125, 127, 131], while there still remain many unexplored functions that can be supported by acoustic signals.

6.2 Acoustic Sensing Frameworks

Acoustic sensing apps are shown to be able to provide several distinct benefits, but their development requires special domain knowledge to address new challenges unseen in traditional apps. For example, handling low-level audio ring buffers might not be straightforward for normal app developers, and it becomes more troublesome when building sensing algorithms on different platforms. For example, setting an Android programming flag, like

`AudioSource.VOICE_RECOGNITION`, can dramatically change the sensing performance. Moreover, different devices usually require substantial fine-tuning, but there is currently no way for developers to *debug* signals easily on different platforms. Many necessary utility functions, like visualizing the sensing signals or setting proper microphone configurations, are also not supported in the state-of-the-art Integrated Development Environments (IDEs), such as Android Studio or XCode.

`LibAS` is a cross-platform framework to solve these unmet needs for developing active acoustic sensing apps. `LibAS` defines a novel abstraction that separates the core sensing algorithms from the platform-dependent controls. This abstraction is designed based on our experience in developing acoustic sensing apps, and has been shown to support many other acoustic sensing apps. `LibAS` has been implemented on several major platforms, such as Android, iOS, Tizen, and Windows/Linux. This cross-platform feature lets developers easily build and test their own acoustic sensing apps on different devices. Note that the cross-platform framework is particularly important for making acoustic sensing apps truly *ubiquitous*.

`LibAS` has been open-sourced and the accompanying DevApp has been published for more than three months. More than three active users have built their own projects with `LibAS`. According to their experiences, `LibAS` is shown to be able to significantly reduce the development effort. By analyzing their code with other similar projects, we observed a reduction of about 80% of lines of code by using `LibAS`. Some users also provide feedback and their expectations for `LibAS`, like a replay feature to test different sensing configurations on the same recording. Many of their suggestions have been reflected in the current `LibAS` and others are part of our future work. We expect that the open-source community can help polish `LibAS` and make it accessible in building many more acoustic sensing apps.

6.3 Limitations and Future Work

Even though we would like to install acoustic sensing apps on all devices/platforms with microphones and speakers, it is still not the case in the real world. For example, during the period when we were trying to implement `EchoTag` on smartwatches, Apple WatchOS 2 didn't support real-time audio recording, Samsung Gear 2 could only record sounds up to 8kHz, and the first generation of Android wear devices didn't have speakers installed. However, as discussed earlier, such a real-time audio support is critical for many apps (not only acoustic sensing), so manufacturers are expected to implement this function in the near future. Today's smartwatches, such as Apple Watch Apple 3 (WatchOS 4), Samsung Gear S3 (Tizen 3.0), and ASUS ZenWatch 4 (Android Wear 2.0), all have such support available. Our applications and frameworks already support many of them, and we are continuously working to support other new platforms. With this trend, it is not difficult to see that more devices that can execute acoustic sensing apps will soon be available and become ubiquitous.

Besides the device/platform limitations, acoustic sensing apps usually need a calibration to compensate for the hardware heterogeneity among different devices. This is an inherent requirement for all cross-device supported applications (just like how apps usually need to dynamically adapt the UI based on screen sizes). We have already built `LibAS` to automatically calibrate some audio properties (like AGC or frequency-selective fading) for acoustic sensing developers, but it is still uncertain if that includes all the calibration that we need for most acoustic sensing apps. An extensive study of hardware heterogeneity is an important prerequisite for future research on acoustic sensing. Specifically, it will be helpful to summarize "critical" properties and collect them by crowdsourcing all existing devices. With this information, acoustic sensing app developers can easily set proper criteria of these critical audio properties, e.g., can sense 15dB via ultrasound band, for their applications. Then, devices which cannot meet this set of requirements will be filtered out automatically when the owners are searching for acoustic apps to install.

Another important issue of acoustic sensing apps is their energy consumption. For example, based on our tests, triggering acoustic sensing on smartphones usually consumes 300mW to 500mW. This result is also consistent with the other energy studies [39]. Since most of the energy is consumed by the speaker/microphones hardware, the signal processing optimization cannot help much to remove it. However, as we have shown, this problem can be mitigated by a proper app design. The key idea is to trigger the acoustic sensing only when necessary. For example, we only enable the acoustic sensing when the accelerometer data seems to be a walking pattern on `BumpAlert`, the phone tilt/WiFi matches a saved fingerprint on `EchoTag`, or the touch event is given by the touch screen for `ForcePhone`. These are akin to how voice assistants (such as Apple Siri or Google Now) enable the microphone recording when certain conditions are met (e.g., the screen has been turned on or the home button is long-pressed). By using these methods, energy consumption can be significantly reduced based on users' usage pattern. Care must be taken for such design considerations in any future acoustic sensing apps.

6.4 Conclusion

In this thesis, we have shown sensing through acoustic signals can be a practical tool for future sensing mechanisms because it is compatible with many existing/future devices. We have proposed and implemented three different acoustic sensing apps on Android and iOS. We have also built an open-source framework to facilitate the development of acoustic sensing apps. Using this tool, we believe many future acoustic sensing apps can be built with minimal development and deployment costs.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Android AutomaticGainControl. https://developer.android.com/reference/android/media/audiofx/AutomaticGainControl.html (archived in Jul 2017: https://web.archive.org/web/20170709024320/https://developer.android.com/reference/android/media/audiofx/AutomaticGainControl.html).

[2] AndroidWalk N Text. https://play.google.com/store/apps/details?id=com.incorporateapps.walktext (archived in Oct 2017: https://web.archive.org/web/20171020061121/https://play.google.com/store/apps/details?id=com.incorporateapps.walktext).

[3] Apple doesn't want you weighing things with your iPhone just yet. http://www.theverge.com/2015/10/28/9625340/iphone-6s-gravity-app-digital-scales (archived in Feb 2017: https://web.archive.org/web/20170205080030/http://www.theverge.com/2015/10/28/9625340/iphone-6s-gravity-app-digital-scales).

[4] Apple Health App. http://www.apple.com/ios/health/ (archived in Nov 2017: https://web.archive.org/web/20171102093105/https://www.apple.com/ios/health/).

[5] Apple iPhone 6s 3D Touch. http://www.apple.com/iphone-6s/3d-touch/ (archived in Sep 2017: https://web.archive.org/web/20170912044616/https://developer.apple.com/ios/3d-touch/).

[6] Apple iPhone 7 Cost Estimations. http://news.ihsmarkit.com/press-release/technology/iphone-7-materials-costs-higher-previous-versions-ihs-markit-teardown-revea (archived in May 2017: https://web.archive.org/web/20170521011516/http://news.ihsmarkit.com/press-release/technology/iphone-7-materials-costs-higher-previous-versions-ihs-markit-teardown-revea).

[7] Apple willing to take hit on iPhone 8 3D Touch sensors that are double the price. http://www.techradar.com/news/apple-willing-to-take-hit-on-iphone-8-3d-touch-sensors-that-are-double-the-price (archived in May 2017: https://web.archive.org/web/20170522110920/http://www.techradar.com/news/apple-willing-to-take-hit-on-iphone-8-3d-touch-sensors-that-are-double-the-price).

[8] Build MEX function from C/C++ or Fortran source code. https://www.mathworks.com/help/matlab/ref/mex.html (archived in Dec 2016: https://web.archive.org/web/20161225193558/http://www.mathworks.com:80/help/matlab/ref/mex.html).

[9] BumpAlert Demo Video. https://youtu.be/X5kj7sFegIY (archived in July 2017: https://kabru.eecs.umich.edu/?attachment_id=1451).

[10] Chinese City Creates a Cell Phone Lane for Walkers. https://http://www.newsweek.com/chinese-city-creates-cell-phone-lane-walkers-271102 (archived in Nov 2016: https://web.archive.org/web/20161117140113/http://www.newsweek.com:80/chinese-city-creates-cell-phone-lane-walkers-271102).

[11] Cocos2d-x: a suite of open-source, cross-platform, game-development tools. http://www.cocos2d-x.org/ (archived in Nov 2017: https://web.archive.org/web/20171101095852/http://www.cocos2d-x.org/).

[12] Ettus Inc., Universal Software Radio Peripheral. http://ettus.com (archived in Oct 2017: https://web.archive.org/web/20171014180904/https://www.ettus.com/).

[13] Fines For Using Smartphones In Crosswalks: Why This Taiwan Law Won't Work. http://www.forbes.com/sites/ralphjennings/2014/05/13/fines-for-using-smartphones-in-crosswalks-why-this-taiwan-law-wont-work/ (archived in Apr 2016: https://web.archive.org/web/20160401023514/http://www.forbes.com:80/sites/ralphjennings/2014/05/13/fines-for-using-smartphones-in-crosswalks-why-this-taiwan-law-wont-work/#54694b897b37).

[14] ForcePhone Demo Video. https://youtu.be/cYxr2wnQVMU (archived in Feb 2018: https://kabru.eecs.umich.edu/?attachment_id=1450).

[15] HTC wants you to squeeze its new phone. https://www.theverge.com/2017/5/16/15643668/htc-wants-you-to-squeeze-its-new-phone (archived in Oct 2017: https://web.archive.org/web/20171017033257/https://www.theverge.com/2017/5/16/15643668/htc-wants-you-to-squeeze-its-new-phone).

[16] Interlink 402 FSR. http://www.interlinkelectronics.com/FSR402.php (archived in Jan 2017: https://web.archive.org/web/20170109203625/http://interlinkelectronics.com/FSR402.php).

[17] LibAcousticSensing Github Repository. https://github.com/yctung/LibAcousticSensing (archived in Nov 2017: https://web.archive.org/web/20171108183308/https://github.com/yctung/LibAcousticSensing).

[18] LibAS Demo: Fingerprint GUI. https://youtu.be/cnep7fFyJhc (archived in Feb 2018: https://kabru.eecs.umich.edu/?attachment_id=1448).

[19] LibAS Demo: Movement Sensing. https://youtu.be/At8imJVRDq4 (archived in Feb 2018: https://kabru.eecs.umich.edu/?attachment_id=1449).

[20] LibAS DevApp in Apple App Store. https://itunes.apple.com/us/app/libas-devapp/id1292387567?ls=1&mt=8 (archived in Nov 2017: https://web.archive.org/save/https://itunes.apple.com/us/app/libas-devapp/id1292387567?ls=1&mt=8).

[21] LibAS DevApp in Google Play Market. https://play.google.com/store/apps/details?id=umich.cse.yctung.devapp (archived in Nov 2017: https://web.archive.org/web/20171108183404/https://play.google.com/store/apps/details?id=umich.cse.yctung.devapp).

[22] MATLAB Coder App. https://www.mathworks.com/products/matlab-coder/apps.html (archived in Nov 2017: https://web.archive.org/web/20171126075919/https://www.mathworks.com/products/matlab-coder/apps.html).

[23] Monsoon Power Monitor. http://www.msoon.com/LabEquipment/PowerMonitor/ (archived in Dec 2016: https://web.archive.org/web/20161225121515/https://www.msoon.com/LabEquipment/PowerMonitor/).

[24] Near Field Communication. http://www.nfc-forum.org (archived in Oct 2017: https://web.archive.org/web/20171025213516/https://nfc-forum.org/).

[25] PhoneGap: build amazing mobile apps powered by open web tech. http://phonegap.com/ (archived in Nov 2017: https://web.archive.org/web/20171106185918/https://phonegap.com/).

[26] Polytec OFV-303 Laser Vibrometer. http://www.polytec.com/us/products/vibration-sensors/ (archived in Oct 2017: https://web.archive.org/web/20171009060418/http://www.polytec.com/us/products/vibration-sensors/).

[27] ReactNative: Learn once, write anywhere: Build mobile apps with React. https://facebook.github.io/react-native/ (archived in Oct 2017: https://web.archive.org/web/20171027163619/https://facebook.github.io/react-native/).

[28] Samsung S Health App. http://shealth.samsung.com/ (archived in Jul 2017: https://web.archive.org/web/20170723112943/http://health.apps.samsung.com/).

[29] Talk When U Walk. https://play.google.com/store/apps/details?id=com.a3logics.talkwyw (archived in Sep 2014: https://web.archive.org/web/20140912014652/https://play.google.com/store/apps/details?id=com.a3logics.talkwyw).

[30] Tesla driver killed in crash with Autopilot active, NHTSA investigating. http://www.theverge.com/2016/6/30/12072408/tesla-autopilot-car-crash-death-autonomous-model-s (archived in Nov 2017: https://web.archive.org/web/20171102011305/https://www.theverge.com/2016/6/30/12072408/tesla-autopilot-car-crash-death-autonomous-model-s).

[31] The joy of generating c code from Matlab. https://www.mathworks.com/company/newsletters/articles/the-joy-of-generating-c-code-from-matlab.html (archived in Aug 2016: https://web.archive.org/web/20160829154327/http://www.mathworks.com:80/company/newsletters/articles/the-joy-of-generating-c-code-from-matlab.html).

[32] Why Apple's iPhone SE lacks 3D Touch technology. http://appleinsider.com/articles/16/03/23/why-apples-iphone-se-lacks-3d-touch-technology (archived in May 2017: https://web.archive.org/web/20170502032338/http://appleinsider.com/articles/16/03/23/why-apples-iphone-se-lacks-3d-touch-technology).

[33] F. Adib, C.-Y. Hsu, H. Mao, D. Katabi, and F. Durand. Capturing the human figure through a wall. *ACM Trans. Graph.*, 34(6):219:1–219:13, Oct. ACM, 2015.

[34] F. Adib, Z. Kabelac, D. Katabi, and R. C. Miller. 3d tracking via body radio reflections. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 317–329. USENIX Association, 2014.

[35] AlDanial. Count Lines of Code. https://github.com/AlDanial/cloc (archived in Dec 2017: https://web.archive.org/web/20171221224627/https://github.com/AlDanial/cloc).

[36] M. T. I. Aumi, S. Gupta, M. Goel, E. Larson, and S. Patel. Doplink: Using the doppler effect for multi-device interaction. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '13, pages 583–586. ACM, 2013.

[37] M. Azizyan, I. Constandache, and R. Roy Choudhury. Surroundsense: Mobile phone localization via ambience fingerprinting. In *Proceedings of the 15th Annual International Conference on Mobile Computing and Networking*, MobiCom '09, pages 261–272. ACM, 2009.

[38] P. Bahl and V. Padmanabhan. Radar: an in-building RF-based user location and tracking system. In *Proceedings of Conference on Computer Communications*, INFOCOM '00, pages 775–784 vol.2. IEEE, 2000.

[39] F. Ben Abdesslem, A. Phillips, and T. Henderson. Less is more: Energy-efficient mobile sensing with senseless. In *Proceedings of the 1st ACM Workshop on Networking, Systems, and Applications for Mobile Handhelds*, MobiHeld '09, pages 61–62. ACM, 2009.

[40] J. Borenstein and Y. Koren. The vector field histogram-fast obstacle avoidance for mobile robots. *Robotics and Automation, IEEE Transactions on*, 7(3):278–288, IEEE, 1991.

[41] J. Brebner. Reaction time in personality theory. *Reaction times*, pages 309–320, Academic Press London, 1980.

[42] R. C. Browning, E. A. Baker, J. A. Herron, and R. Kram. Effects of obesity and sex on the energetic cost and preferred speed of walking. *Journal of Applied Physiology*, 100(2):390–398, Am Physiological Soc, 2006.

[43] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, ACM, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[44] Y. Chen, D. Lymberopoulos, J. Liu, and B. Priyantha. FM-based indoor localization. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 169–182. ACM, 2012.

[45] M.-C. Chiu, S.-P. Chang, Y.-C. Chang, H.-H. Chu, C. C.-H. Chen, F.-H. Hsiao, and J.-C. Ko. Playful bottle: A mobile social persuasion system to motivate healthy water intake. In *Proceedings of the 11th International Conference on Ubiquitous Computing*, UbiComp '09, pages 185–194. ACM, 2009.

[46] J. Chung, M. Donahoe, C. Schmandt, I.-J. Kim, P. Razavai, and M. Wiseman. Indoor location sensing using geo-magnetism. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 141–154. ACM, 2011.

[47] S. Chung and I. Rhee. vtrack: Envisioning a virtual trackpad interface through mm-level sound source localization for mobile interaction. In *Proceedings of the 8th EAI International Conference on Mobile Computing, Applications and Services*, MobiCASE'16, pages 32–41. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2016.

[48] A. R. Derhgawen and D. Ghose. Vision based obstacle detection using 3d hsv histograms. In *Proceedings of Annual IEEE India Conference*, pages 1–4. IEEE, 2011.

[49] S. Elliott. Active control of structure-borne noise. *Journal of Sound and Vibration*, 177(5):651 – 673, Elsevier, 1994.

[50] S. Elmalaki, L. Wanner, and M. Srivastava. Caredroid: Adaptation framework for android context-aware applications. In *Proceedings of the 21st Annual International*

Conference on Mobile Computing and Networking, MobiCom '15, pages 386–399. ACM, 2015.

[51] X. Fan and K. Wong. Migrating user interfaces in native mobile applications: Android to ios. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, pages 210–213. ACM, 2016.

[52] H. Feng, K. Fawaz, and K. G. Shin. Continuous authentication for voice assistants. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, MobiCom '17, pages 343–355. ACM, 2017.

[53] J. Fernandes and J. Neves. Angle invariance for distance measurements using a single camera. In *Industrial Electronics, 2006 IEEE International Symposium on*, volume 1, pages 676–680. IEEE, 2006.

[54] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo. Dsp.ear: Leveraging co-processor support for continuous audio sensing on smartphones. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, SenSys '14, pages 295–309. ACM, 2014.

[55] A. Girouard, J. Lo, M. Riyadh, F. Daliri, A. K. Eady, and J. Pasquero. One-handed bend interactions with deformable smartphones. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 1509–1518. ACM, 2015.

[56] M. Goel, B. Lee, M. T. Islam Aumi, S. Patel, G. Borriello, S. Hibino, and B. Begole. Surfacelink: Using inertial and acoustic sensing to enable multi-device interaction on a surface. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems*, CHI '14, pages 1387–1396. ACM, 2014.

[57] M. Goel, J. Wobbrock, and S. Patel. Gripsense: Using built-in sensors to detect hand posture and pressure on commodity mobile phones. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, UIST '12, pages 545–554. ACM, 2012.

[58] S. Gupta, D. Morris, S. Patel, and D. Tan. Soundwave: Using the doppler effect to sense gestures. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 1911–1914. ACM, 2012.

[59] T. Gustafsson, B. Rao, and M. Trivedi. Source localization in reverberant environments: modeling and statistical analysis. *Speech and Audio Processing, IEEE Transactions on*, pages 791–803, IEEE, 2003.

[60] D. Halperin, W. Hu, A. Sheth, and D. Wetherall. Tool release: Gathering 802.11n traces with channel state information. *SIGCOMM Comput. Commun. Rev.*, 41(1):53–53, Jan. ACM, 2011.

[61] F. J. Harris. On the use of windows for harmonic analysis with the discrete fourier transform. *Proceedings of the IEEE*, 66(1):51–83, IEEE, 1978.

[62] C. Harrison, J. Schwarz, and S. E. Hudson. Tapsense: Enhancing finger interaction on touch surfaces. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 627–636. ACM, 2011.

[63] C. Harrison, D. Tan, and D. Morris. Skinput: Appropriating the body as an input surface. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 453–462. ACM, 2010.

[64] S. Heo and G. Lee. Forcetap: Extending the input vocabulary of mobile touch screens by adding tap gestures. In *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*, MobileHCI '11, pages 113–122. ACM, 2011.

[65] J. D. Hincapié-Ramos and P. Irani. Crashalert: Enhancing peripheral alertness for eyes-busy mobile interaction while walking. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 3385–3388. ACM, 2013.

[66] R. Hooke and J. Yonge. *Lectures de Potentia Restitutiva, Or of Spring Explaining the Power of Springing Bodies*. John Martyn, 1931.

[67] S. Hwang, A. Bianchi, and K.-y. Wohn. Vibpress: Estimating pressure input using vibration absorption on mobile devices. In *Proceedings of the 15th International Conference on Human-computer Interaction with Mobile Devices and Services*, MobileHCI '13, pages 31–34. ACM, 2013.

[68] S. Hwang and K.-y. Wohn. Pseudobutton: Enabling pressure-sensitive interaction by repurposing microphone on mobile device. In *Proceedings of CHI '12 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '12, pages 1565–1570. ACM, 2012.

[69] A. Izquierdo-Fuente, L. del Val, M. I. Jiménez, and J. J. Villacorta. Performance evaluation of a biometric system based on acoustic images. In *Proceedings of Sensors*, volume 11, pages 9499–9519. Molecular Diversity Preservation International, 2011.

[70] S. Jain, C. Borgiattino, Y. Ren, M. Gruteser, Y. Chen, and C. F. Chiasserini. Lookup: Enabling pedestrian safety services via shoe sensing. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pages 257–271. ACM, 2015.

[71] T. Kaler, J. P. Lynch, T. Peng, L. Ravindranath, A. Thiagarajan, H. Balakrishnan, and S. Madden. Code in the air: Simplifying sensing on smartphones. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 407–408. ACM, 2010.

[72] N. Karapanos, C. Marforio, C. Soriente, and S. Čapkun. Sound-proof: Usable two-factor authentication based on ambient sound. In *Proceedings of the 24th USENIX*

Conference on Security Symposium, SEC'15, pages 483–498. USENIX Association, 2015.

[73] A. Khattab, J. Camp, C. Hunter, P. Murphy, A. Sabharwal, and E. W. Knightly. Warp: A flexible platform for clean-slate wireless medium access protocol design. *SIGMOBILE Mob. Comput. Commun. Rev.*, 12(1):56–58, Jan. ACM, 2008.

[74] Y.-H. Kim. Sound propagation: An impedance based approach. Wiley, 2010.

[75] M. Kunita. Range measurement in ultrasound fmcw system. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 90(1):9–19, Wiley, 2007.

[76] K. Kunze and P. Lukowicz. Symbolic object localization through active sampling of acceleration and sound signatures. In *Proceedings of the 9th International Conference on Ubiquitous Computing*, UbiComp '07, pages 163–180. Springer-Verlag, 2007.

[77] Y.-S. Kuo, P. Pannuto, K.-J. Hsiao, and P. Dutta. Luxapose: Indoor positioning with mobile phones and visible light. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*, MobiCom '14, pages 447–458. ACM, 2014.

[78] N. D. Lane, P. Georgiev, and L. Qendro. Deepear: Robust smartphone audio sensing in unconstrained acoustic environments using deep learning. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, pages 283–294. ACM, 2015.

[79] G. Laput, E. Brockmeyer, S. E. Hudson, and C. Harrison. Acoustruments: Passive, acoustically-driven, interactive controls for handheld devices. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 2161–2170. ACM, 2015.

[80] G. Laput, X. A. Chen, and C. Harrison. Sweepsense: Ad hoc configuration sensing using reflected swept-frequency ultrasonics. In *Proceedings of the 21st International Conference on Intelligent User Interfaces*, IUI '16, pages 332–335. ACM, 2016.

[81] P. Lazik and A. Rowe. Indoor pseudo-ranging of mobile devices using ultrasonic chirps. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, SenSys '12, pages 391–392. ACM, 2012.

[82] H. Lee, T. H. Kim, J. W. Choi, and S. Choi. Chirp signal-based aerial acoustic communication for smart devices. In *Proceedings of Conference on Computer Communications*, INFOCOM '15, pages 2407–2415. IEEE, 2015.

[83] F. Li, H. Chen, X. Song, Q. Zhang, Y. Li, and Y. Wang. Condiosense: High-quality context-aware service for audio sensing system via active sonar. *Personal Ubiquitous Comput.*, 21(1):17–29, Feb. Springer-Verlag, 2017.

[84] Q. Li, J. long Han, and D. jun Wu. Survey on predicting and controlling of structure-borne noise from rail transit bridges. In *Proceedings of Electric Technology and Civil Engineering (ICETCE)*, pages 4559–4563, 2011.

[85] J. Lim, A. Amado, L. Sheehan, and R. E. Van Emmerik. Dual task interference during walking: The effects of texting on situational awareness and gait stability. *Gait & posture*, 42(4):466–471, Elsevier, 2015.

[86] K. Liu, X. Liu, and X. Li. Guoguo: Enabling fine-grained indoor localization via smartphone. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 235–248. ACM, 2013.

[87] S. Low, Y. Sugiura, D. Lo, and M. Inami. Pressure detection on mobile phone by camera and flash. In *Proceedings of the 5th Augmented Human International Conference*, AH '14, pages 11:1–11:4. ACM, 2014.

[88] H. Lu, W. Pan, N. D. Lane, T. Choudhury, and A. T. Campbell. Soundsense: Scalable sound sensing for people-centric applications on mobile phones. In *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services*, MobiSys '09, pages 165–178. ACM, 2009.

[89] W. Mao, J. He, H. Zheng, Z. Zhang, and L. Qiu. High-precision acoustic motion tracking: Demo. In *Proceedings of the 22Nd Annual International Conference on Mobile Computing and Networking*, MobiCom '16, pages 491–492. ACM, 2016.

[90] P. Marti and I. Iacono. Evaluating the experience of use of a squeezable interface. In *Proceedings of the 11th Biannual Conference on Italian SIGCHI Chapter*, CHItaly '15, pages 42–49. ACM, 2015.

[91] J. Minguez. The obstacle-restriction method for robot obstacle avoidance in difficult environments. In *Proceedings of IROS*, pages 2284–2290. IEEE, 2005.

[92] R. Nandakumar, K. K. Chintalapudi, V. Padmanabhan, and R. Venkatesan. Dhwani: Secure peer-to-peer acoustic nfc. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 63–74. ACM, 2013.

[93] R. Nandakumar, S. Gollakota, and N. Watson. Contactless sleep apnea detection on smartphones. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pages 45–57. ACM, 2015.

[94] R. Nandakumar, V. Iyer, D. Tan, and S. Gollakota. Fingerio: Using active sonar for fine-grained finger tracking. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 1515–1525. ACM, 2016.

[95] J. L. Nasar and D. Troyer. Pedestrian injuries due to mobile phone use in public places. *Accident Analysis & Prevention*, 57(0):91 – 95, Elsevier, 2013.

[96] S. Nirjon, R. F. Dickerson, P. Asare, Q. Li, D. Hong, J. A. Stankovic, P. Hu, G. Shen, and X. Jiang. Auditeur: A mobile-cloud service platform for acoustic event detection on smartphones. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 403–416. ACM, 2013.

[97] M. Ono, B. Shizuki, and J. Tanaka. Touch & activate: Adding interactivity to existing objects using active acoustic sensing. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, pages 31–40. ACM, 2013.

[98] M. Ono, B. Shizuki, and J. Tanaka. Sensing touch force using active acoustic sensing. In *Proceedings of the Ninth International Conference on Tangible, Embedded, and Embodied Interaction*, TEI '15, pages 355–358. ACM, 2015.

[99] E. W. Pedersen and K. Hornbæk. Expressive touch: Studying tapping force on tabletops. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 421–430. ACM, 2014.

[100] C. Peng, G. Shen, Y. Zhang, Y. Li, and K. Tan. Beepbeep: A high accuracy acoustic ranging system using cots mobile devices. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, pages 1–14. ACM, 2007.

[101] V. Philomin, R. Duraiswami, and L. Davis. Pedestrian tracking from a moving vehicle. In *Intelligent Vehicles Symposium, 2000. IV 2000. Proceedings of the IEEE*, pages 350–355, 2000.

[102] C. J. Plack. The sense of hearing. Lawrence Erlbaum Associates, Inc., 2005.

[103] N. B. Priyantha, A. Chakraborty, and H. Balakrishnan. The cricket location-support system. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, MobiCom '00, pages 32–43. ACM, 2000.

[104] Q. Pu, S. Gupta, S. Gollakota, and S. Patel. Whole-home gesture recognition using wireless signals. In *Proceedings of the 19th Annual International Conference on Mobile Computing & Networking*, MobiCom '13, pages 27–38. ACM, 2013.

[105] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: Enabling interactive perception applications on mobile devices. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 43–56. ACM, 2011.

[106] G. Ramos, M. Boulos, and R. Balakrishnan. Pressure widgets. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 487–494. ACM, 2004.

[107] L. Ravindranath, A. Thiagarajan, H. Balakrishnan, and S. Madden. Code in the air: Simplifying sensing and coordination tasks on smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, HotMobile '12, pages 4:1–4:6. ACM, 2012.

[108] J. Rosen and L. Q. Gothard. *Encyclopedia of Physical Science (Facts on File Science Library), Volume 1 & 2*. Facts on File, 2010.

[109] M. Rossi, J. Seiter, O. Amft, S. Buchmeier, and G. Tröster. Roomsense: An indoor positioning system for smartphones using active sound probing. In *Proceedings of the 4th Augmented Human International Conference*, AH '13, pages 89–95. ACM, 2013.

[110] W. Ruan, Q. Z. Sheng, L. Yang, T. Gu, P. Xu, and L. Shangguan. Audiogest: Enabling fine-grained hand gesture detection by decoding echo signal. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '16, pages 474–485. ACM, 2016.

[111] A. Sahami Shirazi, N. Henze, T. Dingler, K. Kunze, and A. Schmidt. Upright or sideways?: Analysis of smartphone postures in the wild. In *Proceedings of the 15th International Conference on Human-computer Interaction with Mobile Devices and Services*, MobileHCI '13, pages 362–371. ACM, 2013.

[112] S. Salemian, H. Keivani, and O. Mahdiyar. Comparison of radar pulse compression techniques. In *Proceedings of International Symposium on Microwave, Antenna, Propagation and EMC Technologies for Wireless Communications*, volume 2, pages 1076–1079 Vol. 2. IEEE, 2005.

[113] S. Sen, R. R. Choudhury, and S. Nelakuditi. Spinloc: Spin once to know your location. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, HotMobile '12, pages 12:1–12:6. ACM, 2012.

[114] H. Shuldiner. Volvo stops for pedestrians. *Ward's Dealer Business*, 43(12):9, 12 2009.

[115] Z. Sun, A. Purohit, R. Bose, and P. Zhang. Spartacus: Spatially-aware interaction for mobile devices through energy-efficient audio sensing. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 263–276. ACM, 2013.

[116] S. P. Tarzia, P. A. Dinda, R. P. Dick, and G. Memik. Indoor localization without infrastructure using the acoustic background spectrum. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 155–168. ACM, 2011.

[117] D. M. J. Tax and P. Laskov. Online svm learning: from classification to data description and back. In *2003 IEEE XIII Workshop on Neural Networks for Signal Processing (IEEE Cat. No.03TH8718)*, pages 499–508. IEEE, 2003.

[118] Y.-C. Tung and K. G. Shin. Echotag: Accurate infrastructure-free indoor location tagging with smartphones. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15, pages 525–536. ACM, 2015.

[119] Y.-C. Tung and K. G. Shin. Expansion of human-phone interface by sensing structure-borne sound propagation. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, pages 277–289. ACM, 2016.

[120] Y. C. Tung and K. G. Shin. Use of phone sensors to enhance distracted pedestrians's safety. *IEEE Transactions on Mobile Computing*, PP(99):1–1, IEEE, 2017.

[121] I. Ulrich and I. R. Nourbakhsh. Appearance-based obstacle detection with monocular color vision. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 866–871. AAAI Press, 2000.

[122] V. N. Vapnik. *Statistical Learning Theory*. Wiley-Interscience, 1998.

[123] H. Veeraraghavan, O. Masoud, and N. Papanikolopoulos. Computer vision algorithms for intersection monitoring. *Intelligent Transportation Systems, IEEE Transactions on*, 4(2):78–89, IEEE, 2003.

[124] J. Wang, K. Zhao, X. Zhang, and C. Peng. Ubiquitous keyboard for small mobile devices: Harnessing multipath fading for fine-grained keystroke localization. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 14–27. ACM, 2014.

[125] Q. Wang, K. Ren, M. Zhou, T. Lei, D. Koutsonikolas, and L. Su. Messages behind the sound: Real-time hidden acoustic signal capture with smartphones. In *Proceedings of the 22Nd Annual International Conference on Mobile Computing and Networking*, MobiCom '16, pages 29–41. ACM, 2016.

[126] T. Wang, G. Cardone, A. Corradi, L. Torresani, and A. T. Campbell. Walksafe: A pedestrian safety app for mobile phone users who walk and talk while crossing roads. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, HotMobile '12, pages 5:1–5:6. ACM, 2012.

[127] W. Wang, A. X. Liu, and K. Sun. Device-free gesture tracking using acoustic signals. In *Proceedings of the 22Nd Annual International Conference on Mobile Computing and Networking*, MobiCom '16, pages 82–94. ACM, 2016.

[128] J. Xiong and K. Jamieson. Towards fine-grained radio-based indoor location. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, HotMobile '12, pages 13:1–13:6. ACM, 2012.

[129] M. Youssef and A. Agrawala. The horus wlan location determination system. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, MobiSys '05, pages 205–218. ACM, 2005.

[130] S. Yun, Y.-C. Chen, and L. Qiu. Turning a mobile device into a mouse in the air. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pages 15–29. ACM, 2015.

[131] H. Zhang, W. Du, P. Zhou, M. Li, and P. Mohapatra. Dopenc: Acoustic-based encounter profiling using smartphones. In *Proceedings of the 22Nd Annual International Conference on Mobile Computing and Networking*, MobiCom '16, pages 294–307. ACM, 2016.

[132] Z. Zhang, D. Chu, X. Chen, and T. Moscibroda. Swordfight: Enabling a new class of phone-to-phone action games on commodity phones. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 1–14. ACM, 2012.

[133] Z. Zhang, X. Zhou, W. Zhang, Y. Zhang, G. Wang, B. Y. Zhao, and H. Zheng. I am the antenna: Accurate outdoor ap location using smartphones. In *Proceedings of the 17th Annual International Conference on Mobile Computing and Networking*, MobiCom '11, pages 109–120. ACM, 2011.

[134] H. Zheng, J. Mou, W. Lin, and E. Ong. Modeling and prediction of structure-borne seek noise of hard disk drives. *Magnetics, IEEE Transactions on*, 45(11):4933–4936, IEEE, 2009.

[135] P. Zhou, Y. Zheng, Z. Li, M. Li, and G. Shen. Iodetector: A generic service for indoor outdoor detection. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, SenSys '12, pages 361–362. ACM, 2012.

[136] Z. Zhou, W. Diao, X. Liu, and K. Zhang. Acoustic fingerprinting revisited: Generate stable device id stealthily with inaudible sound. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 429–440. ACM, 2014.

[137] X. Zhu, Q. Li, and G. Chen. Apt: Accurate outdoor pedestrian tracking with smartphones. In *Proceedings of Conference on Computer Communications*, INFOCOM '13, pages 2508–2516. IEEE, 2013.