Cross-Platform Support for Rapid Development of Mobile Acoustic Sensing Applications

Yu-Chih Tung
The University of Michigan
yctung@umich.edu

Duc Bui
The University of Michigan
ducbui@umich.edu

Kang G. Shin
The University of Michigan
kgshin@umich.edu

ABSTRACT

LibAS is a cross-platform framework to facilitate the rapid development of mobile acoustic sensing apps. It helps developers quickly realize their ideas by using a high-level Matlab script, and test them on various OS platforms, such as Android, iOS, Tizen, and Linux/Win. LibAS simplifies the development of acoustic sensing apps by hiding the platform-dependent details. For example, developers need not learn Objective-C/SWIFT or the audio buffer management in the CoreAudio framework when they want to implement acoustic sensing algorithms on an iPhone. Instead, developers only need to decide on the sensing signals and the callback function to handle each repetition of sensing signals. We have implemented apps covering three major acoustic sensing categories to demonstrate the benefits and simplicity of developing apps with LibAS. Our evaluation results show the adaptability of LibAS in supporting various acoustic sensing apps and tuning/improving their performance efficiently. Developers have reported that LibAS saves them a significant amount of time/effort and can reduce up to 90% lines of code in their acoustic sensing apps.

CCS CONCEPTS

• **Human-centered computing** → **Ubiquitous and mobile computing systems and tools**; • **Software and its engineering** → **Application specific development environments**;

KEYWORDS

Acoustic sensing, cross-platform development, rapid prototype

ACM Reference Format:
Yu-Chih Tung, Duc Bui, and Kang G. Shin. 2018. Cross-Platform Support for Rapid Development of Mobile Acoustic Sensing Applications. In *MobiSys '18: The 16th Annual International Conference on Mobile Systems, Applications, and Services, June 10–15, 2018, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3210240.3210312

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 [12, 26, 32, 33, 38, 44, 49], extend human–computer interfaces [14, 20, 21, 28, 37, 50, 52], and create nearly zero-cost inter-device interactions [11, 19, 30, 35, 47, 56, 57]. 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.

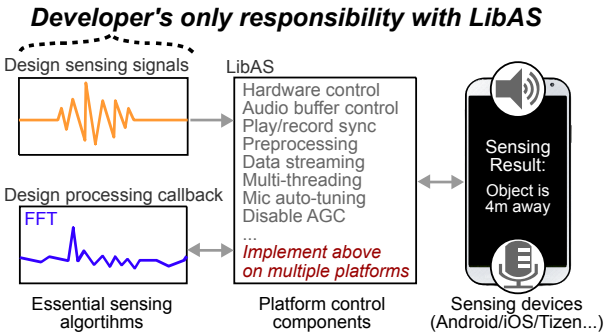


Figure 1: Concept of LibAS. LibAS reduces the cross-platform development effort for acoustic sensing apps by hiding laborious platform-dependent programming details.

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 JavaScript on Android and Objective-C/SWIFT on iOS, are difficult, time-consuming, and error-prone [16]. Moreover, any modification of the algorithms causes changes of code on multiple platforms, entailing significant development and maintenance efforts/costs.

Platform settings. Since each platform requires specific settings, developers must have detailed knowledge to set different platforms correctly. For example, adding a recorder flag in Android (`AudioSource.VOICE_RECOGNITION`) might change the acoustic sensing behavior significantly 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

MobiSys '18, June 10–15, 2018, Munich, Germany

ACM ISBN 978-1-4503-5720-3/18/06…$15.00
https://doi.org/10.1145/3210240.3210312

Sonar-like sensing		Inter-device interactions		Sound fingerprinting	
System	Testbed	System	Testbed	System	Testbed
FingerIO[37]	Galaxy S4 & wearables	PseudoRanging[30]	Special mic and amp	Acoustruments[28]	iPhone 5c
SoundWave[20]	MAC Air & 3 laptops	DopEnc[56]	6 Android devices	Symbolic Location[26]	Nokia 5500 Sport
ApneaApp[36]	4 Android devices	SwordFight[57]	Nexus One & Focus	Touch & Activate[39]	Arduino
vTrack[14]	Galaxy Note4 & S2	BeepBeep[40]	HP iPAQ & Dopod 838	EchoTag[49]	4 Android & 1 iPhone
AudioGest[45]	Galaxy S4/Tab & Mac	Spartacus[47]	Galaxy Nexus	RoomSense[44]	Galaxy S2
LLAP[54]	Galaxy S5	AAMouse[55]	Google Nexus 4 & Dell laptop	CondioSense[32]	4 Android devices
BumpAlert[51]	6 Android devices	Dhwani[35]	Galaxy S2 & Hp mini	SweepSense[29]	Mac & earbuds

Table 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.

the algorithms to different device hardware. For example, some microphones might even receive 20dB less signal strength at 22kHz than others [30]. 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. In particular, 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. For example, several researchers [31, 40, 49, 57] mentioned the problem caused by non-real-time OS delay and then solved the problem by syncing the played/recorded sound through their own ad-hoc solutions. These repeated efforts can be avoided or significantly reduced by providing a proper abstraction that handles the common platform issues systematically.

Fig. 1 shows the concept of LibAS which divides a normal acoustic sensing app into 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 5 platforms: 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 support cross-platform dvelopment). 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 first use the remote mode 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-first-then-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 [2, 7, 8], but we are the first to apply it to acoustic sensing apps.

Our evaluation results show that LibAS 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 incurred 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 3];
- Implementation of three existing apps from different categories using LibAS [Section 4 / Section 5];
- Evaluation of the overhead of LibAS and its capability to handle the effects of heterogeneous hardware/platform support [Section 6]; and
- User study of three developers who used LibAS in their real-world projects [Section 7].

2 RELATED WORK

Table 1 summarizes the existing acoustic sensing apps. Acoustic sensing is expected to become ubiquitous as it can be integrated into

[1]LibAS Github: https://github.com/yctung/LibAcousticSensing

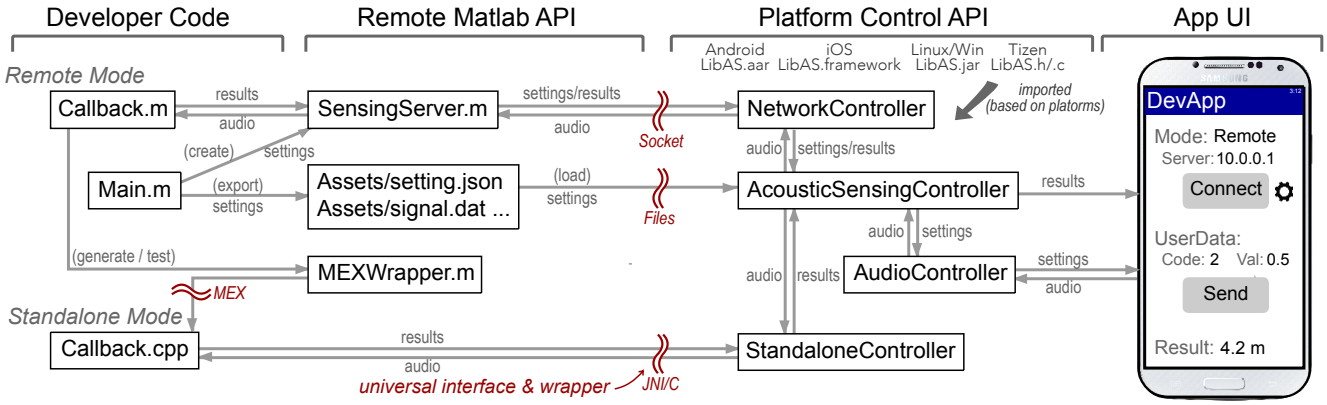


Figure 2: System overview. LibAS provides a universal interface/wrapper to communicate with the callback components. Thus, the platform control API can be easily imported to support different devices/platforms while keeping the developer's essential sensing algorithm consistent.

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 [12, 21, 22, 33, 44, 48, 52]. 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 [14, 28, 39, 55], determine the relative movements between devices [30, 40, 47, 56, 57], remember the tagged locations [26, 44, 49], or recognize the users' status or their interactions with devices [20, 36, 37, 50]. LibAS is designed mainly for active acoustic sensing but it can also be used for passive acoustic sensing. Although acoustic sensing apps are usually claimed to be *ubiquitous*, most of them are actually implemented and tested only on one or two types of devices (as shown in Table 1). 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 [30], lack of device microphone controls [52], speaker ringing effects [35], significant physical separation of microphones from speakers [50], random speaker jitters [57], and adaptation of microphone automatic gain control (AGC) [49]. 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 [18] is a system-wide support to accelerate the acoustic sensing computation by utilizing phone's GPU while DeepEar [27] is a library for constructing a convolution neural network (CNN) based on acoustic sensing signals. Auditeur [38] and SoundSense [33] focus on providing a scalable/cloud-based service to collect acoustic features. CAreDroid [15] is an Android-only framework to provide an efficient sensing (including acoustic signals) interface for context-aware computations. In terms of providing a cross-platform abstraction for smartphone sensors, the closest to LibAS is Code In The Air (CITA) [23, 42]. However, CITA focuses on *tasking apps*, such as sending a message to users' wives when they leave office, and provides 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 [7], ReactNative [8], and Cocoa2d-x [2] frameworks where developers can build cross-platform mobile app/game UI by JavaScript or C++.

3 SYSTEM DESIGN

Fig. 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.

3.1 Design Challenges

Developing acoustic sensing apps across different platforms is inherently challenging because of the fragmented programming language, as we mentioned in Section 1. Designing a framework like LibAS to facilitate such a cross-platform development requires even more challenges to overcome. First, we need a programming abstraction to provide the cross-platform support. It is difficult to decide on a proper abstraction due to the trade-off between development overhead and programmability. For example, an extreme design choice may implement all existing acoustic sensing apps as black boxes across different platforms, and app developers can then utilize these black boxes in a way similar to `AcousticSensing.onTowDevicesNearby(range, callback)`. This way, developers may minimize the development overhead but lose the flexibility of developing new acoustic sensing algorithms (because of "inflexible" black boxes) and improving the sensing performance (which usually requires additional signal processing). LibAS's abstractions are designed on the basis of our experience with, and study of more than 20 existing acoustic sensing apps. Our current design, as shown in Fig. 2, allows developers to preserve the programmability for customizing sensing signals and processing the responses while hiding tedious platform details.

Second, since acoustic sensing needs extensive tuning, developers usually need to iteratively modify their algorithms based on the observed performance. It is thus critical to visualize acoustic responses and then adjust the sensing algorithms accordingly. To achieve this, LibAS provides not only the standalone mode, which supports sensing algorithms written in C to be executed on different platforms, but also a Matlab remote mode that allows developers to prototype the system using a high-level script language.

Finally, while LibAS is designed for many existing acoustic sensing apps, our current design may still miss critical information/-functions which will be needed in future acoustic sensing apps. So, it is important to make LibAS extensible to meet developers' special needs. Currently, LibAS has an extensible user interface that allows developers to communicate additional data between their sensing algorithms and devices. For example, one of the app developers — who are using LibAS— utilized this interface to send accelerometer data for improving the acoustic sensing accuracy. The usage of this interface will be elaborated in Section 5

3.2 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 preprocessed 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 functionality 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 [12, 12, 26, 31, 34, 36, 47, 49–51, 55].

3.3 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 [6, 9]. 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 detection 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. 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.

3.4 Expected Development Flow

Fig. 3 shows a typical development flow of using LibAS. New developers may first install our remote Matlab package and the pre-built

[2] MEX [1] is an interface to call native C functions in Matlab

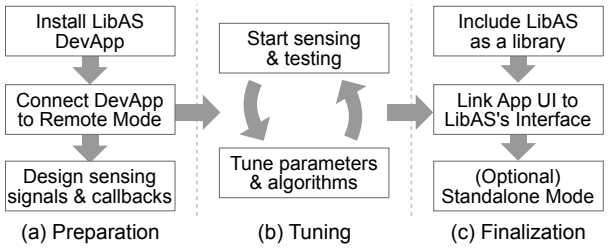


Figure 3: Expected development flow. Developers can first use the published LibAS DevApp (cross-platform supported) to realize their idea without even installing platform development kits, like XCode or AndroidStudio.

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. 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 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 [6, 9], 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.

3.5 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. 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 [2], also helps build cross-platform games in C. Our remote mode model is also similar to several popular cross-platform UI frameworks like PhoneGap [7] and ReactNative [8] 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.

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 not trivial 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 [12, 26, 47, 49, 50, 55], 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 6.

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 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.

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

```
function [ret] = SonarCallback(context, action,
    data, user)
  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 = SenisngResultDoubleArray(dists);

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

Code 2: SonarCallback.m. The remote callback focuses on processing each repetition of the sensing signals received.

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 [43]. 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.

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 [46] (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 1 (main script) and Code 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 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. This `SensingServer` is the main primitive used in Matlab to communicate and control devices via the `AcousticSensingController` over different platforms. A few other constants necessary for the callback to parse the received signal — e.g., the signal to correlate as the matched filter in this example — can be assigned to a global variable called `PS`.

As the remote callback function shown in Code 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 [36, 37, 45, 49–51], 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 are omitted due to space limit.

Since the recorded signal has already been synchronized and segmented by LibAS, the processing of each recorded chirp is straightforward. As shown in Code 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

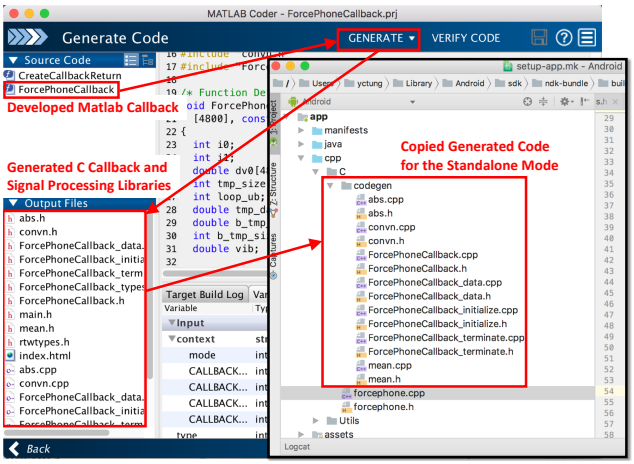


Figure 4: Standalone callback created by Matlab Coder. The C-based standalone callback can be automatically generated based on algorithm developed and tested in the Matlab remote mode.

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 [6, 9], as the example shown in Fig. 4). 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. 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. See Section 7 for details of user experience of `LibAS`.

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 [55], create a new gesture control between smartphones [47], and detect if two users are walking toward each other [56]. Our demo app implemented with `LibAS` can be viewed as a generalization of these movement sensing apps.

Based on Doppler's theory [43], the relationship between the changed movement speed and the shifted central frequency can be expressed as: $f_c' - f_c = vf_c/c$, 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

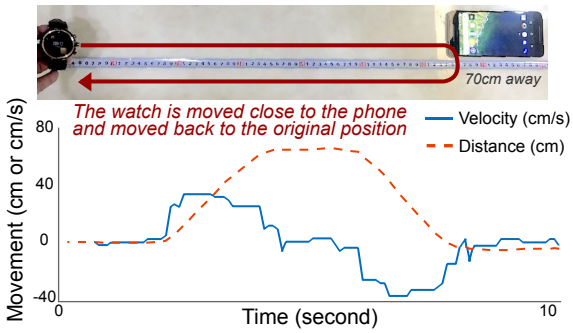


Figure 5: Movement sensing by Doppler shifts. The integrated area of velocity indicates the movement shift. A demo video can be found at `https://youtu.be/At8imJVRDq4`

overlapped sampling to further improve the sensing resolution and response latency [47, 55, 56]. 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 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 [30, 31, 35, 53]. A video of this demo app can be found at `https://youtu.be/At8imJVRDq4` [4].

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

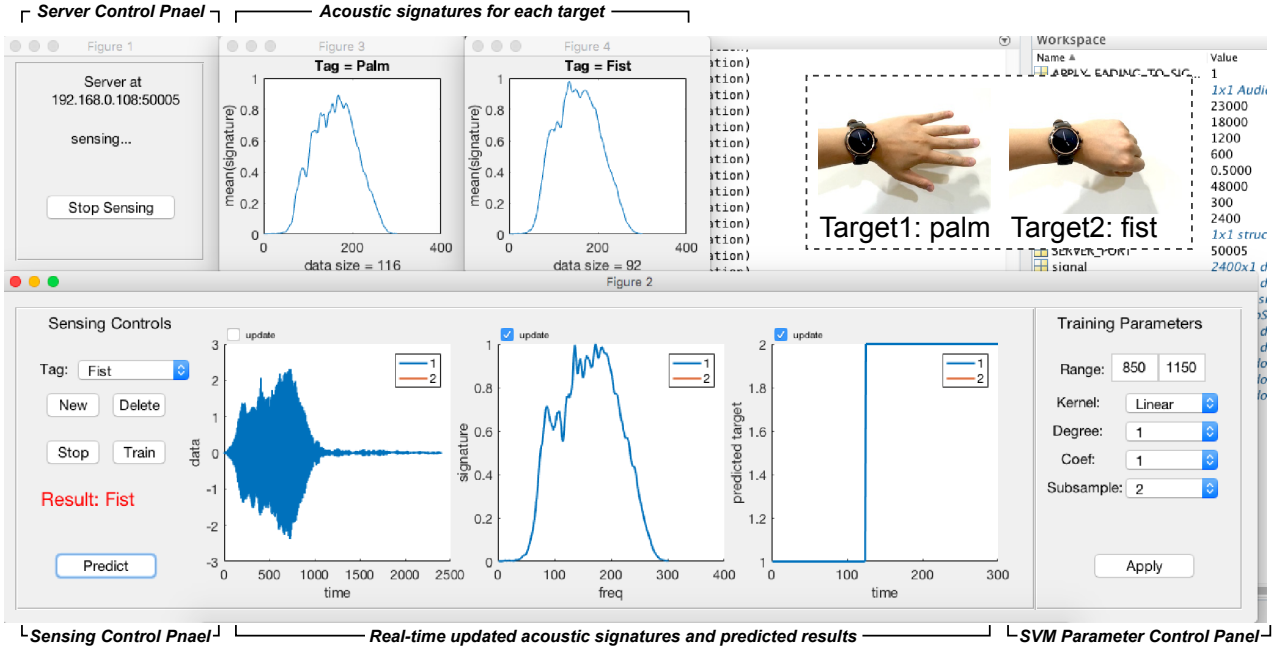


Figure 6: Graphical User Interface (GUI) for fingerprinting acoustic signatures. Developers can easily classify different user-defined actions based on acoustic signatures. A demo video of this GUI can be found at `https://youtu.be/cnep7fFyJhc`

can be used to distinguish rooms [44], remember the tagged locations [49], recognize the touched surface [26], and sense how the phone is touched [39]. 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. 6. 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 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://youtu.be/cnep7fFyJhc` [3].

This demo app shows how the GUI-support of Matlab and the capability of integrating 3rd-party libraries, i.e., LibSVM [13], can help develop acoustic sensing apps using LibAS. We have also used this GUI to realize other fingerprinting apps, such as EchoTag [49], 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 [48] and others [12, 33], which use the sound of environments as acoustic signatures, rather than the sensing signals sent by the device.

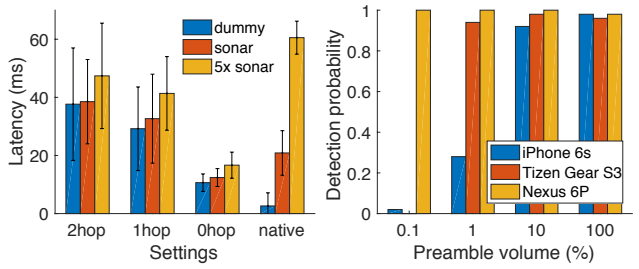
6 EVALUATION

We will first evaluate the overhead of our current LibAS implementation, in terms of the response latency and the pre-processing 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.

6.1 Overhead

As 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, however, there can be an additional overhead due to network connection.

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. 7(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



(a) Response latency (b) Preamble detection

Figure 7: Overheads. The small overhead of LibAS can support most real-time acoustic sensing apps.

to the same 802.11n WiFi router), (ii) 1-hop (the sensing device is connected 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 and are thus omitted (due to the space limit).

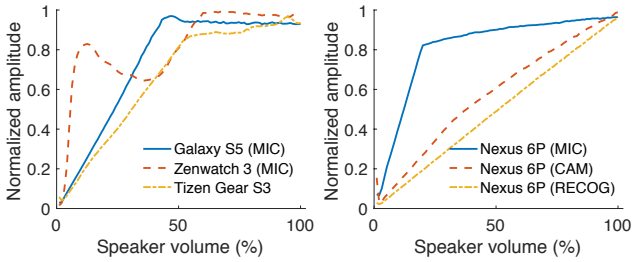
As shown in Fig. 7(a), without considering the processing cost of the sensing algorithm, (i.e., a dummy case), the remote mode of LibAS can achieve an average response latency less than 40ms in a common 2-hop setting. This latency can be reduced further to 30ms by turning either the laptop or phone into a hotspot and connecting it 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 in the same machine, the response latency can be reduced further to 10ms. By closely anatomizing the delay, 6ms is found to come from the way our Java socket interface is hooked to Matlab while only the other 4ms is spent for 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 practically impossible to provide a real-time response. Even though our standalone mode can push the response latency to be 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. The low latency overhead of LibAS can meet the requirement of many real-time acoustic sensing apps, such as the sonar demo app that needs the processing to be done in 50ms (i.e., before sending the next 2400-sample signals sampled at 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 7(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 [41] 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 whichever mode fits best to their purpose. As we will discuss in Section 7, the minimal/small latency overhead of LibAS meets the real-time requirements of our current users.

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 design similar to existing approaches [31, 49, 50]. 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 [46]. 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.

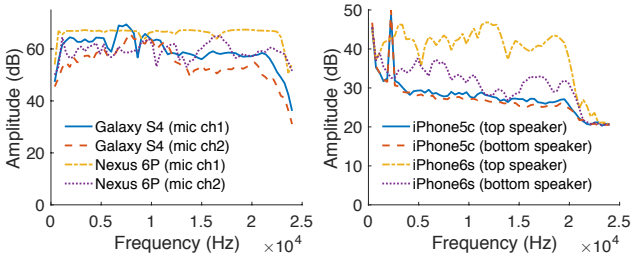
Our current preamble detections have been experimented on more than 20 devices. For the acoustic sensing apps that sense the signal sent by itself, such as our sonar demo app, LibAS can easily achieve higher than a 98% detection probability when 20% of the maximum volume is used to play the preamble. 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 7(b), this performance might vary with devices 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 subsections.

Note that this 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. 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



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

Figure 8: Automatic gain control detections (AGC). LibAS detects if AGC is enabled by sending a signal with linearly increased volumes.



(a) Different microphones (b) Different speakers

Figure 9: Frequency responses of various devices. The sensed frequency responses vary not only with devices but also with the microphone/speaker used to sense.

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.

6.2 Adaptability

Adaptability is an important performance metric of LibAS since it is designed to support various apps, platforms, and devices. Our demo apps have shown that LibAS's design is general enough to support several categories of acoustic sensing algorithms. In what follows, we will focus on how LibAS can adapt itself to different platforms/devices to improve the sensing performance and help developers make the correct sensing configurations. (Other real-world use-experience of LibAS by our current developers will be discussed in the next subsection.)

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. 8 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. 8(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 the session mode to `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. 8(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).

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. 9 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 [30], but only the microphone gains were reported.

As shown in Fig. 9, 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 at 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. 9(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.

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 [50]. 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.

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 [25]. 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.

7.2 Benefits of Using LibAS

The biggest common benefit reported by all three users was the visualization provided by LibAS's remote mode. 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.

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 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.

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 [51] 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 2 shows the lines of code estimated with the open-sourced cloc tool [10], 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 pre-built 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.

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

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 [7] or ReactNative [8]. Some other features — that one may want to see — include the support of Apple's Watch OS4 and Tizen TV, more demo examples, and a crowdsourcing platform to share developers' sensing configurations.

LibAS is designed mainly for real-time *active* acoustic sensing apps, such as the sonar-like sensing example mentioned earlier. LibAS could also support "non-sensing" apps, such as real-time sound masking or authentication based on acoustic signal processing [17, 24]. It is used to implement a demo to send data over devices by inaudible chirps [5]. By setting a dummy null signal to send, LibAS can also be used for *passive* acoustic sensing apps, i.e., sensing by monitoring the recorded environment noises [48]. However, there is not much room to reduce platform-dependent code if apps need not play the sensing signals actively. Current LibAS supports only those apps that send fixed and repeated sensing signals (which is the common case for most acoustic sensing projects). We chose such a design to simplify the abstraction. Adding advanced APIs to dynamically change the sensing signals is part of our future work.

LibAS has been open-sourced for several months and also used by at least 6 developers. 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, enabling LibAS to help acoustic sensing apps become truly ubiquitous.

9 CONCLUSION

In this paper, we have presented LibAS, a cross-platform framework to ease the development of acoustic sensing apps. Developing cross-platform 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.

10 ACKNOWLEDGEMENTS

The authors would like to thank Arun Ganesan, Timothy Trippel, and Kassem Fawaz of RTCL at University of Michigan, the current users of LibAS, the anonymous reviewers, and the shepherd for constructive comments on the earlier versions of this paper. The work reported in this paper was supported in part by NSF under Grant CNS-1646130.

REFERENCES

[1] Build MEX function from C/C++ source code. https://www.mathworks.com/help/matlab/ref/mex.html.

[2] Cocos2d-x: a suite of open-source, cross-platform, game-development tools. http://www.cocos2d-x.org/.

[3] LibAS Demo: Fingerprint GUI. https://youtu.be/cnep7fFyJhc.

[4] LibAS Demo: Movement Sensing. https://youtu.be/At8imJVRDq4.

[5] LibAS Example: Chirp Messenger. https://github.com/yctung/LibAcousticSensing/tree/master/Example/ChirpMessenger.

[6] MATLAB Coder App. https://www.mathworks.com/products/matlab-coder/apps.html.

[7] PhoneGap: build amazing mobile apps powered by open web tech. http://phonegap.com/.

[8] ReactNative: Learn once, write anywhere: Build mobile apps with React. https://facebook.github.io/react-native/.

[9] The joy of generating c code from Matlab. https://www.mathworks.com/company/newsletters/articles/the-joy-of-generating-c-code-from-matlab.html.

[10] AlDanial. Count Lines of Code. https://github.com/AlDanial/cloc.

[11] 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 ACM UbiComp '13*, pages 583–586.

[12] M. Azizyan, I. Constandache, and R. Roy Choudhury. Surroundsense: Mobile phone localization via ambience fingerprinting. In *Proceedings of ACM MobiCom '09*, pages 261–272.

[13] 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, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[14] S. Chung and I. Rhee. vtrack: Virtual trackpad interface using mm-level sound source localization for mobile interaction. In *Proceedings of ACM UbiComp '16*, pages 41–44.

[15] S. Elmalaki, L. Wanner, and M. Srivastava. Caredroid: Adaptation framework for android context-aware applications. In *Proceedings of ACM MobiCom '15*, pages 386–399.

[16] X. Fan and K. Wong. Migrating user interfaces in native mobile applications: Android to ios. In *Proceedings of MOBILESoft '16*, pages 210–213.

[17] H. Feng, K. Fawaz, and K. G. Shin. Continuous authentication for voice assistants. In *Proceedings of ACM MobiCom'17*, pages 343–355.

[18] 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 ACM SenSys '14*, pages 295–309.

[19] 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 ACM CHI '14*, pages 1387–1396.

[20] S. Gupta, D. Morris, S. Patel, and D. Tan. Soundwave: Using the doppler effect to sense gestures. In *Proceedings of ACM CHI '12*, pages 1911–1914.

[21] C. Harrison, J. Schwarz, and S. E. Hudson. Tapsense: Enhancing finger interaction on touch surfaces. In *Proceedings of ACM UIST '11*, pages 627–636.

[22] C. Harrison, D. Tan, and D. Morris. Skinput: Appropriating the body as an input surface. In *Proceedings of ACM CHI '10*, pages 453–462.

[23] 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 ACM SenSys '10*, pages 407–408.

[24] 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.

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

[26] K. Kunze and P. Lukowicz. Symbolic object localization through active sampling of acceleration and sound signatures. In *Proceedings of UbiComp '07*, pages 163–180.

[27] N. D. Lane, P. Georgiev, and L. Qendro. Deepear: Robust smartphone audio sensing in unconstrained acoustic environments using deep learning. In *Proceedings of ACM UbiComp '15*, pages 283–294.

[28] G. Laput, E. Brockmeyer, S. E. Hudson, and C. Harrison. Acoustruments: Passive, acoustically-driven, interactive controls for handheld devices. In *Proceedings of ACM CHI '15*, pages 2161–2170.

[29] 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, 2016.

[30] P. Lazik and A. Rowe. Indoor pseudo-ranging of mobile devices using ultrasonic chirps. In *Proceedings of ACM SenSys '12*, pages 99–112.

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

[32] 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. 2017.

[33] 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 ACM MobiSys '09*, pages 165–178.

[34] W. Mao, J. He, and L. Qiu. Cat: High-precision acoustic motion tracking. In *Proceedings of ACM MobiCom '16*, pages 69–81.

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

[36] R. Nandakumar, S. Gollakota, and N. Watson. Contactless sleep apnea detection on smartphones. In *Proceedings of ACM MobiSys '15*, pages 45–57.

[37] R. Nandakumar, V. Iyer, D. Tan, and S. Gollakota. Fingerio: Using active sonar for fine-grained finger tracking. In *Proceedings of ACM CHI '16*, pages 1515–1525.

[38] 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 ACM MobiSys '13*, pages 403–416.

[39] M. Ono, B. Shizuki, and J. Tanaka. Touch & activate: Adding interactivity to existing objects using active acoustic sensing. In *Proceedings of ACM UIST '13*, pages 31–40.

[40] 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 ACM SenSys '07*, pages 1–14.

[41] 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 ACM MobiSys '11*, pages 43–56.

[42] L. Ravindranath, A. Thiagarajan, H. Balakrishnan, and S. Madden. Code in the air: Simplifying sensing and coordination tasks on smartphones. In *Proceedings of HotMobile '12*, pages 4:1–4:6.

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

[44] 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 ACM AH '13*, pages 89–95.

[45] 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 ACM UbiComp '16*, pages 474–485.

[46] S. Salemian, M. Jamshihi, and A. Rafiee. Radar pulse compression techniques. In *Proceedings of WSEAS AEE'05*, pages 203–209.

[47] Z. Sun, A. Purohit, R. Bose, and P. Zhang. Spartacus: Spatially-aware interaction for mobile devices through energy-efficient audio sensing. In *Proceeding of ACM MobiSys '13*, pages 263–276.

[48] S. P. Tarzia, P. A. Dinda, R. P. Dick, and G. Memik. Indoor localization without infrastructure using the acoustic background spectrum. In *Proceedings of ACM MobiSys '11*, pages 155–168.

[49] Y.-C. Tung and K. G. Shin. Echotag: Accurate infrastructure-free indoor location tagging with smartphones. In *Proceedings of MobiCom '15*, pages 525–536.

[50] Y.-C. Tung and K. G. Shin. Expansion of human-phone interface by sensing structure-borne sound propagation. In *Proceedings of ACM MobiSys '16*, pages 277–289.

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

[52] 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 ACM MobiSys '14*, pages 14–27.

[53] 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 ACM MobiCom '16*, pages 29–41.

[54] W. Wang, A. X. Liu, and K. Sun. Device-free gesture tracking using acoustic signals. In *Proceedings of ACM MobiCom '16*, pages 82–94.

[55] S. Yun, Y.-C. Chen, and L. Qiu. Turning a mobile device into a mouse in the air. In *Proceedings of ACM MobiSys '15*, pages 15–29.

[56] H. Zhang, W. Du, P. Zhou, M. Li, and P. Mohapatra. Dopenc: Acoustic-based encounter profiling using smartphones. In *Proceedings of ACM MobiCom '16*, pages 294–307.

[57] 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 ACM MobiSys '12*, pages 1–14.