A Smart Framework for Fine-Grained Microphone Acoustic Permission Management

Weili Han, *Member, IEEE*, Chang Cao, Zhe Zhou, Shize Chen, Lingqi Huang, and X. Sean Wang, *Senior Member, IEEE*

Abstract—Microphones attracted a lot of attentions from attackers due to the sensitivity of voice data: attackers may control devices through abusing their microphones, fingerprint devices by measuring their microphones, or directly monitor the microphone readings to steal users' private data. Nevertheless, OS developers failed to address the severe consequences. While the current security mechanism only offers a coarse-grained access control over the usage of microphones: recording all sound or shutting off, it is necessary to redesign the microphone security mechanism to enforce fine-grained restrictions over the usage of microphones. In this article, we propose a fine-grained microphone access control scheme on Android platform, referred to as *FMC* (Finer Microphone Controller). In our scheme, microphone acoustic permissions are granted with three finer policies: *treble policy*, *timbre policy* and *exclusion policy*, with which most of the attacks mentioned above can be defended against. In addition, to ease user's policy management, we employ a smart policy recommendation method, avoiding additional manual policy approvals. The results in our experiments show that a negligible 1.06 percent performance overhead is incurred during policy enforcement. Besides, the policy recommendation system in *FMC* promises an accuracy of 82.82 percent averagely. We believe that our work is a practical defense scheme against attacks exploiting microphone acoustic permissions and should be employed by OS developers.

Index Terms—Access control, android, sensing control, microphone, permission management

1 Introduction

Nowadays, microphones have become an indispensable part of nearly every mobile device to support various services such as voice assistants and sound-based payment, bringing convenience to our daily life. To support these services, apps are widely using microphones. *WeChat*, for example, uses microphones to support voice messaging for billions of users. According to a survey posted by *Pew Research Center* [1] in 2014, 6.11 percent out of one million apps on *Google Play Store* are requesting the microphone permission `RECORD_AUDIO`.

However, the existing access control and permission management mechanism in mobile devices only enforces a coarse-grained option for users: recording all sound or shutting off. Several research efforts have shown that such a mechanism can be exploited by attackers to steal users' private data. For example, unrestricted microphone access may help attackers generate fingerprints [2], [3], [4], which can be further used to track devices and thereby their owners. Combined with data from other onboard sensors, real-time recorded audio data help attackers extract sensitive information (e.g., keystrokes, PINs, and environment information) [5], [6]. Besides, with the imperfections in acoustic hardware and the perceptual differences between human ears and sensors, attackers can even construct an inaudible and hidden communication channel [7], [8].

Two issues in the current microphone acoustic permission management framework make mobile devices susceptible for attacks: i) unrestricted audio carry-on information accessing; ii) unrestricted concurrent access to acoustic functionalities. In the current Android audio management framework, recorded audio information is fully provided to an app if the recording permission `RECORD_AUDIO` is granted. Here, no filter is applied to the audio data before these data are presented to the app. Therefore, sensitive information which can be deeply analyzed from audio should be managed finely. Next, the current framework neglects the potential risks from the concurrent work of acoustic components. For instance, the audio covert channel [9] can be set up via abusing speakers and microphones simultaneously. This attack implies that the speaker, which is not covered in the current Android permission mechanism, should be finely controlled.

To address the above-mentioned security risks for mobile devices, prior works try to improve the current sensor management framework by carrying out a fine-grained access control [10], [11]. However, researchers have rarely focused on the physical and concurrent features of acoustic components. The previous mechanisms usually control the audio data as a whole according to access context. In addition, the adoption of their schemes has extra overhead for users since few smart user management solutions or recommendations have been raised [12], [13].

In this paper, we propose a fine-grained and smart microphone access control scheme, referred to as Finer Microphone

- *W. Han, C. Cao, S. Chen, L. Huang, and X.S. Wang are with Software School, and Shanghai Key Laboratory of Data Science, Fudan University, Shanghai 200433, China. E-mail: {wlhan, 16212010001, 17212010054, 16302010054, xywangCS}@fudan.edu.cn.*
- *Z. Zhou is with the School of Computer Science, Fudan University, Shanghai 200433, China. E-mail: zhouzhe@fudan.edu.cn.*

Manuscript received 13 Aug. 2018; revised 5 Nov. 2019; accepted 19 Dec. 2019. Date of publication 0 . 0000; date of current version 0 . 0000.
(Corresponding author: Weili Han.)
Digital Object Identifier no. 10.1109/TDSC.2019.2962403

Controller (*FMC*) and implement its prototype on Android platform. By defining three finer permissions over mobile devices' audio management and enforcing their corresponding control policies, the usages of acoustic components in mobile devices are effectively restricted. More specifically, sensitive information carried in audio can optionally be wiped out and the simultaneous use of speakers and microphones is restricted under *FMC*'s control. As a result, its fine-grained policies can effectively mitigate the microphone related attacks [2], [3], [4], [5], [6], [8], [14] to varying degrees. Moreover, *FMC* imposes little extra overhead on users as it provides smart control recommendations of our proposed finer permissions for users.

We summarize our contributions in this paper as follows:

- We thoroughly explore the attack surface rooted from the embedded microphone acoustic permission mechanism and find three weak points that require fine-grained permission protection: i) unrestricted access to the high frequency channel, ii) residual fingerprint within audio, and iii) simultaneous usage of acoustic components.
- To avoid unnecessarily leaking sensitive data (including high frequency wave and fingerprint) to apps, we design the *treble policy* and *timbre policy* to restrict the use of high frequency channel and fingerprints respectively. Specifically, we add a high-frequency filter to enforce the *treble policy* and an acoustic feature eraser to enforce the *timbre policy*.
- Considering the possible severe consequences brought by the simultaneous use of microphones and speakers, we apply a Dynamic Separation of Duty (DSoD) policy over microphones and speakers to the Android audio management framework to enforce the *exclusion policy*.
- We design a policy recommendation framework for the smart recommendation of the three new policies mentioned above. We also implement a prototype, referred to as Finer Microphone Controller, and apply it on Android devices.

To validate the effectiveness and measure the overhead of our proposed *FMC*, we conduct a series of experiments, including overall performance evaluation and a series of operational delay tests. The evaluation results show that *FMC* has only a 1.06 percent performance overhead. In addition, according to the evaluation experiments on compatibility and functional effectiveness, we find that neither crashes nor errors occurred during the whole evaluation. Thus, *FMC* offers a smooth and secure user experience. Finally, our evaluation of 33,972 apps from *Google Play* shows that the current third-party app markets require a fine-grained and smart microphone permission management framework to a large extent.

Road Map. The rest of paper is organized as follows: Section 2 introduces the background and motivation; Section 3 explains adversaries addressed in our work and presents our threat model; Section 4 presents the design of *FMC*; Section 5 demonstrates the implementation of the key modules in *FMC*, in which the policy enforcement mainly lies in the native layer of Android, then effectively prevents our codes from being bypassed or tampered; Section 6 evaluates *FMC* for its performance, compatibility and effectiveness; Section 7 empirically studies the popular app market *Google Play Store* to investigate the microphone permission declaration in the real world; Section 8 discusses the limitations of *FMC* and introduces our future work; Section 9 overviews the related works; Finally, Section 10 summarizes our work.

2 Background and Motivation

Most of the sensor-based attacks, especially voice-based attacks, viciously exploit the inherent vulnerabilities within either acoustic features or operating systems. In this section, we present a brief introduction about acoustic features, onboard sensors and Android permission administration. Then we thoroughly analyze the issues of the current Android permission mechanism, which motivates our research.

Acoustic Features. There are three physical acoustic features of the human voice:

- *Volume*, *a.k.a.* loudness, is positively correlated to the power of the signal. It is also referred to as the energy intensity of audio signals.
- *Pitch*, which represents the frequency, is a perceptual property of sounds. It reflects the speed of vibration.
- *Timbre*, which is characterized by the waveform within a clip of audio signal, represents the feature of the sound generator. It can be used to distinguish different instruments or different people.

The pitch range varies with different speakers. Human ears have a limited audible range, which is from about 20 Hz to 20,000 Hz for a healthy adult. But the frequency range beyond 20,000 Hz can also carry abundant information. Several apps use these parts precisely to develop their functions (e.g., *Alipay*'s soundwave payment uses both low-frequency range and high-frequency range to make a payment). However, the mobile phones' surplus range, beyond the necessary requirements of apps, also becomes the key to various inaudible attacks. That is, the inaudible audio can be exploited to extract sensitive information or initiate other severe attacks [8], [15]. Besides, the individual difference in high frequency range is enough for attackers to fingerprint both the user and the device [4].

A series of features and indices are proposed in recent years to better characterize voice signals for all kinds of purposes, such as 1) *RMS*, which stands for the square root of the arithmetic mean of the squares of the signal strength at various frequencies, 2) *Mel-Frequency Cepstrum Coefficient* (*MFCC*), which is a spectral derived index, and 3) *Spectral Entropy*, which mainly depends on the peaks of a spectrum and their locations. Among these features and indices, *MFCC* is one of the most widely used features in speech recognition and has been proved to be superior in fingerprinting acoustic sources, including smartphones [3]. *MFCCs* are the coefficients that collectively make up a mel-frequency cepstrum (*MFC*). Essentially, *MFCCs* of a signal are a small set of features which concisely describe the overall shape of a spectral envelope, and they are often used to describe the timbre. That is, destroying the *MFCC* of a piece of audio data means an alteration of the timbre. Then the audio would not be recognized as from the same source anymore.

Acoustic Components in Mobile Devices. On a mainstream mobile phone, there can even be two or more microphones

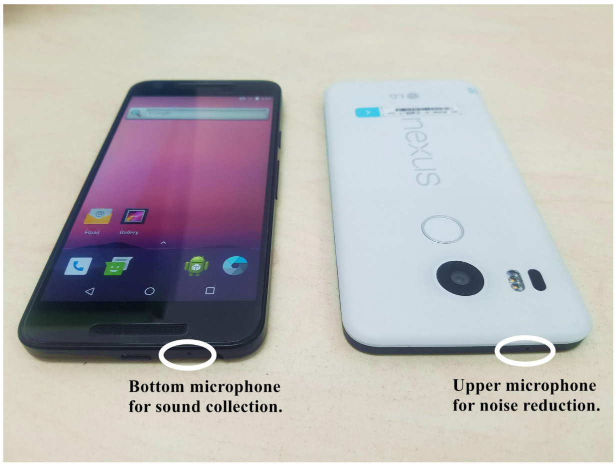


Fig. 1. Nexus 5X devices used in our work. Each device embeds more than one microphones.

for different purposes. For example, Nexus 5X places two microphones at the top and the bottom of the device respectively, as shown in Fig. 1. Generally, the microphone on the top is used for denoising, while the bottom one is primarily used for recording. In Android, apps can record via AudioRecord API to receive Pulse-Code Modulation (PCM) format audio data. There are mainly three parameters for this set of APIs: sound channel, sampling bit, and sampling rate. The sound channel can be set as mono or stereo. The sampling bit refers to how much storage will be used to store one sample. Basically, the sampling bit can be set as 8 bits or 16 bits. The sampling rate, refers to as the number of audio samples gathered in a second, is usually set as 16 kHz or 44.1 kHz. In general, the quality of an audio file increases while the sampling rate increases, and the sound file restores the sound with higher fidelity. Microphones absorb sound waves and output analog electronic signals. Analog to digital conversion circuit will then digitalize the analog signal, and its output can be recognized by digital circuits.

The speaker in an Android device is not covered in Android permission framework. That is, there is no permission prerequisite for apps to use the speaker. It is reasonable when we consider the speaker as an information output component only. However, the concurrent use of microphones and speakers may result in a security hazard. In the work [16], a speaker-microphone device fingerprinting method has been proposed to uniquely identify the devices. Note that, *FMC* proposed in this paper is motivated to put forward a novel policy, *exclusion policy*, to finely control the sensitive access to the speaker.

In a word, these acoustic components on phones not only make it easy for the apps to collect data, but also bring the risks of leaking data.

Sensor Permission Administration on Android. An Android app, which runs in a limited-access sandbox, has to request corresponding permissions if it needs to access resources or information outside its own sandbox. Every app contains an AndroidManifest.xml file in its installer root directory to declare all the permissions required. From Android 6.0 (API level 23) on, a subset of permissions can be dynamically revoked instead of all being fixed after the installation [17]. Onboard sensors, including cameras, microphones and GPS are all protected with such permission restrictions, while major standard sensors (i.e., accelerometer, gyroscope, pressure, gravity) are not protected by Android permission mechanism. Permissions on Android platform are divided into four basic types of security level, i.e., *normal*, *dangerous*, *signature*, and *signatureOrSystem*, among which *normal* and *dangerous* are the two most common types. The microphone permission belongs to *dangerous* permissions, in which case an Android device will prompt users to approve or reject the request explicitly at runtime. In Android, an app must have RECORD_AUDIO permission to legally access audio sensors. Once it gets the permission, there are two sets of APIs to access audio sensors: AudioRecord, which directly provides a raw sound stream for the app; and MediaRecord, which offers a compressed audio file.

With the introduction of Android runtime permissions, users now have better control over permissions than before, because they can freely turn on or turn off any single permission at any time. However, the coarse granularity [18] of the RECORD_AUDIO permission is still problematic. Namely, the microphone permission can only be set to either on or off, while the carried-on information like environment signals cannot be separately reserved or wiped out.

3 Threat Model

We consider third-party apps that use microphones as the adversaries in our threat model. The adversaries already have the permission to use the embedded microphones on the hosting phones. Users would not revoke the permission even if they can do so with the help of runtime permission, because the users need the adversarial apps to complete their claimed legitimate functionalities related to microphones, like online chatting and voice assistant.

The adversaries assumed in this paper are curious about the auxiliary sensitive information carried by voice data, including but not limited to the location, the identity of a device, and the identity of a user. These data are obliviously sensitive and valuable for the adversaries. Meanwhile, it is difficult to obtain these data, because they are usually protected by the corresponding permissions. For instance, a device's ID can only be accessed with READ_PHONE_STATE permission. Similarly, the location information should be accessed with ACCESS_FINE_LOCATION or ACCESS_COARSE_LOCATION permission.

However, using the audio data, the adversaries can acquire sensitive information without the designated permissions. The reason is that the information carried by audio is far richer than apps' actual needs, and the current audio management does not sanitize the data before handing them over to apps. For example, via recording APIs, a voice chat app can get from both the audio's high-frequency components and the background noise surrounding its user, even though the information is irrelevant with and would not be helpful to the claimed function of the app.

We categorize the existing microphone-related attacks via third-party apps into three classes as follows.

- Adversaries trying to infer or steal sensitive information (e.g., location, keystrokes) from the high frequency audio channel.

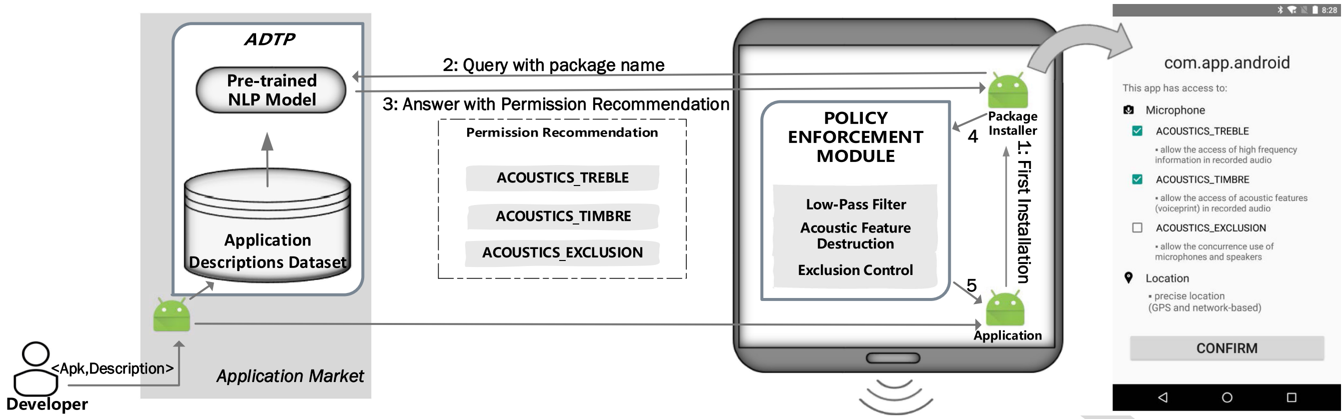


Fig. 2. The design of *FMC* framework with the interface of modified *Package Installer* (right) to check and confirm new sub-permissions. The recommended sub-permissions would be checked first and the user can reselect as his or her preferences.

- Adversaries trying to uniquely identify devices or even users. Through device identification, the adversaries can thereby mark or even track the devices. These attacks mainly exploit the acoustic features carried by audio data collected through microphones [2], [3], [4].
- Adversaries trying to inject abnormal signals into the microphone recordings. The injected data can even be utilized to take control of the devices. Such injections can be finished in an inaudible way through the high frequency channel [8], [19].

Note that, some of the attacks mentioned above can be accomplished through the collaboration of microphone and other onboard sensors on smart devices. Our work only considers threats resulted from microphone acoustic permission management, while other sensors could also be protected following the same routine.

To eliminate the threats above, our work introduces three fine-grained sub-permissions to restrict apps' microphone usage. Moreover, a permission recommendation method is proposed to help users intelligently set such sub-permissions, relieving users of the policy administration burdens.

4 *FMC*: Design

We design *FMC* in order to mitigate the voice-based attacks in mobile devices. The basic idea of our design is to split the current coarse `RECORD_AUDIO` permission into three sub-permissions to counter the mentioned threats. Further, we provide a smart recommendation service to reduce administration burdens of users. The overall design is built on the current third-party app market's client-server model.

As shown in Fig. 2, the framework of *FMC* consists of two main parts: (1) An Acoustic Description-to-Permission (*ADTP*) inference module, which is used to help app markets automatically recommend acoustic sub-permissions of apps to users. For apps uploaded to third-party app markets like *Google Play Store*, developers are required to submit a brief description of their apps to explain the apps' main functions and to provide security-related information like the permissions required by the apps. This module leverages *Natural Language Processing (NLP)* technologies to analyze the description for each app and work out their actual required level of information from the audio data. Accordingly, the module can decide if an app requires the three sub-permissions we propose. The output of the module is three bits in our implementation, indicating whether each of the three sub-permissions should be granted to the app. (2) A Policy Enforcement Module, which enforces sensitive data sanitization and microphone-speaker concurrent use management system-wide. This module is embedded at the native layer of Android audio framework. It receives the output from *ADTP* and user's adjustment if necessary and accordingly enforces the policies for each app.

When a new app is uploaded to the app market, it is added into the Application Description Dataset. *ADTP* usually finishes its training beforehand (the training can be triggered manually by the manager or can be set automatically) and stores a pre-trained NLP model at the market side. As shown in Fig. 2, from a client side perspective, a complete request-solution flow is as follows:

1) The user downloads an app from the app market and initiates an installation. An embedded application *Package Installer* is responsible for the installation.
2) Once *Package Installer* realizes that a new app is to be installed, it queries the market side with the package name of the app.
3) *ADTP* at the market side receives the query and looks for the description of the app. Furthermore, it provides a permission recommendation as the answer to *Package Installer*.
4) *Package Installer* shows the recommendation to the user while the user can decide whether to adjust the decision on each sub-permission. The final permission selections would be passed to the Policy Enforcement Module.
5) Finally, the Policy Enforcement Module starts working, enforcing corresponding policies when the app runs.

In our design, we use *Package Installer* as a control center, and it is responsible for transferring messages. The two-step interaction between client and server is fast and efficient. Note that, here, *FMC* shows a basic and key process to intelligently and finely control microphone-relevant operations. When Android apps update their versions, *FMC* would keep the previous settings or recommend new settings. The current design can be extended to accommodate for the new requirements.

4.1 Fine-Grained Acoustic Permissions

As mentioned, the threats from third-party apps are mainly due to the surplus range of frequency, the excess of acoustic features in the audio data, and unlimited concurrent usage of acoustic components in mobile devices. Inspired by the acoustic features and the existing categories of attacks pointed out in Section 3, we propose three sub-permissions for audio usages to supplement the original `RECORD_AUDIO`:

- *ACOUSTICS_TREBLE* represents if the app is allowed to use the high-frequency components of the audio. This sub-permission prevents apps from freely using the high frequency channel, which could be further exploited to get the location, identifier, or other sensitive information. When this sub-permission is not granted, we enforce the *treble policy* which filters out the high frequency channel.
- *ACOUSTICS_TIMBRE* represents if the app can access the acoustic fingerprints either of the device or of the user inside the audio. This sub-permission prevents adversaries from identifying or tracking devices or their owners. This sub-permission is designed to enforce the *timbre policy* which destructs the acoustic features contained in the audio data.
- *ACOUSTICS_EXCLUSION* represents if the app can use microphones and speakers simultaneously. This sub-permission prevents apps from generating acoustic fingerprints of the phone or injecting voice commands to the phone. This sub-permission is designed to enforce the *exclusion policy* which limits the concurrent usage of microphones and speakers.

These three sub-permissions can be judged and enforced independently, effectively countering each of the threats mentioned in Section 3. Note that, only when `RECORD_AUDIO` permission is granted, will the sub-permissions take effect to limit the privilege of `RECORD_AUDIO` permission.

4.2 *ADTP*: Permission Filtering and Matching

The goal of *ADTP* is to work out whether each of the sub-permissions proposed in Section 4.1 should be granted to each app according to their descriptions. Here, we use *Long Short Term Memory Networks* (LSTM) [20], a variant of *RNN* which is capable of learning long-term dependencies, to help translate the descriptions into permissions. We choose *LSTM* here because of its superior performance on classification and prediction problems. The module is assumed to be running at the server side, when an app with its description is uploaded to a market.

ADTP stores a pre-trained NLP model and uses it to analyze the description to figure out the necessity of the three sub-permissions. The module not only extracts the keywords inside the description, which can directly reflect the requirements of permissions, but also analyzes the description semantics in order to get a higher precision in prediction. For example, when the word *microphone* does not appear explicitly in the description, but the words like *voice chatting* or *online chatting* are found, the framework would recommend using microphones. Under this circumstance, all three sub-permissions would be granted to preserve user experiences: the microphone and speaker are supposed to be working at the same time to support instant communication, and any acoustic features of the caller should not be destroyed in order to provide a regular listening experience.

4.3 Policy Enforcement

The Policy Enforcement Module in *FMC* is designed to restrict the usage of audio collection and speaker according to the settings of the sub-permissions proposed in Section 4.1.

The Policy Enforcement Module acquires the permission list from the server through a query started by *Package Installer* when a user installs an app. Then the following three policy enforcement elements are applied according to the acquired permission list when *FMC* serves the app with audio data.

Treble Policy: Low-Pass Filtering. If the sub-permission `ACOUSTICS_TREBLE` is not granted, the audio data are sanitized with a low-pass filter before handed over to apps. This policy enforcement ensures that sensitive information on the high frequency channel is wiped out, while the enforcement has minimal impact on user experiences. Usually, apps need only audible components, while inaudible components like high frequency components are not required.

This low-pass filter directly prevents adversaries from launching attacks via the high frequency channel. For example, those apps trying to listen at the high frequency covert channel can no longer receive information delivered by the transmitters.

Timbre Policy: Acoustic Feature Destruction. Acoustic features, including *MFCC*, are usually studied as the key to fingerprinting devices as well as users. In order to protect users from being identified or tracked, when the sub-permission `ACOUSTICS_TIMBRE` is not granted, we need to destroy the acoustic features contained in the audio data. Note that, the human readable information in the audio data should be preserved. Since *MFCC* is the *de facto* acoustic feature in fingerprinting, the *timbre policy* enforcement aims to destroy it without sacrificing the functionalities of apps.

Exclusion Policy: Permission Exclusion Control. The current Android system does not provide a mechanism to restrict the concurrent use of different acoustic components. However, it has been confirmed that the concurrent use of a microphone with a speaker can result in severe attacks [4], [9]. Therefore, our framework is designed to restrict the work of the speaker when the microphone is being used by an app with the sub-permission `ACOUSTICS_EXCLUSION` revoked. Note that, in this paper, we leverage the concept of Separation of Duty (SoD) [21], [22] to mitigate the risk resulted from the collusion between the usages of microphones and speaker. To be more specific, the *exclusion policy* reflects the principle of Dynamic Separation of Duty, which means that a subject cannot simultaneously activate or use two sensitive permissions although they have been granted both at that time.

Exception. The Policy Enforcement Module also provides an adjustment mechanism for users. That is, users can override the recommendation of *ADTP* at both installation and run time.

5 *FMC*: Implementation of Key Modules

In this section, we explain how the key modules of our prototype on Android are implemented. We leverage NLP

TABLE 1
Application Category Composition in Google Play Store Dataset

Category	Number	Category	Number
ART&DESIGN	50	LIFESTYLE	2,485
AUTO&VEHICLES	113	MAPS&NAVIGATION	329
BEAUTY	51	MEDICAL	796
BOOKS&REFERENCE	612	MUSIC&AUDIO	3,504
BUSINESS	3,562	NEWS&MAGAZINES	613
COMICS	51	PARENTING	65
COMMUNICATION	2,388	PERSONALIZATION	356
DATING	96	PHOTOGRAPHY	794
EDUCATION	3,598	PRODUCTIVITY	1,492
ENTERTAINMENT	2,420	SHOPPING	407
EVENTS	67	SOCIAL	1,150
FINANCE	749	TOOLS	2,275
FOOD&DRINK	219	TRAVEL&LOCAL	1,154
GAMES	2,798	VIDEO PLAYERS&EDITORS	720
HEALTH&FITNESS	886	WEATHER	31
HOUSE&HOME	81	LIBRARIES&DEMO	60

These 33,972 apps from 32 categories are downloaded in July, 2018.

technologies to implement *ADTP*. In addition, low-pass filter, human acoustic fingerprint interference and exclusion control of acoustic components are implemented to enforce our proposed acoustic policies.

5.1 *ADTP* Module

When we build *ADTP*, we employ Sequence Semantic Embedding[1] (SSE), which is a TensorFlow based encoder framework toolkit for NLP related tasks and is now a commercial text classification toolkit, to do Description-to-Permission translation. Benefiting from TensorFlow's convenient deep learning blocks like LSTM, SSE can easily support large scale NLP related machine learning tasks, including text classification.

In order to train an effective permission classification model, we need to collect and label app description data. Thus, we first collect a large number (615,781 apps) of description samples from *Google Play Store* in July, 2018. After translating all non-English descriptions (8,532 apps) into English using *Google Translate* and filtering out non-English characters, special Unicode symbols, and blanks in the descriptions, we gather 33,972 samples, which request the microphone permission in their `AndroidManifest.xml` file. They fall into 32 categories as shown in Table 1.

Next, we label all the samples on whether they entail demands for the three sub-permissions. The labels are generated based on keywords and validated by us. The keyword-based labeling process can be summarized as follows: for each triplet of our proposed policy composition, we first scan all the description texts, and search for a pre-defined set of keywords. We label each app using a triplet of bits, each representing one policy, where*1* means it should be enforced to restrict the app and *0* means otherwise. For example, the keywords for policy composition *000* include *voice chat*, *talk to*, *video chat* and even some more complex phrases like *high-quality recording*, *communicate with the parent*, *emergency call*. For instance, *Shazam*, an app for instant music identification and discovery, is labeled as *001*, which stands for implementing *exclusion policy* only, because we find *identify music* and *find new music* as keywords in its description, indicating that *Shazam* may need the microphone when it is trying to identify the played music. In this case, the speaker is not required. Meanwhile, as a music recognition app, in order to identify a song correctly, *Shazam* should preserve the full frequency information and the human voice contained in the music played around. Therefore, we do not restrict its use by enforcing the *treble policy* or the *timbre policy*. Conversely, *Arabic To English: Voice & Text Translation Free*, a translation app in our data set, is recognized and labeled as *110*, since keywords such as *voice translator*, *text to speech* and *voice recognition* are found. Utility tools such as translators can function normally without timbre information of the speaker as well as treble information, since the frequencies of human voice are mainly distributed in a low-frequency interval.

After keyword recognition, peer volunteers check all labels together with the descriptions manually and adjust the labels according to their experiences.

In order to cross validate the model's prediction accuracy, we apply a 10-fold cross validation on the model. After training the *ADTP* model using the SSE's dual-encoder model on the training set, we validate the model using the validation set. The three bits are evaluated as a whole, which means only when all three bits are correctly predicted, do we judge it as a true-positive sample. The average validation accuracy of ten rounds is 82.82 percent averagely, with a variance of 4.96×10^{-5}.

Note that, in this paper, the *ADTP* model is trained for our proposed sub-permissions based on the data only from *Google Play Store*. Hence, we cannot guarantee that the performance of our current model is the same for app descriptions from some other app markets. However, the methodology itself is general and can be easily extended to other markets.

5.2 Policy Enforcement Module

As shown in Fig. 2, the Policy Enforcement Module consists of three sub-modules: 1) a low-pass filter for high frequency acoustic information filtering, 2) human acoustic fingerprint interference, and 3) exclusion control of acoustic components.

When a user downloads and installs an app from an app market, *Package Installer* will then obtain the recommended configuration of the sub-permissions from *ADTP* in *FMC*. Then, the recommended sub-permissions configuration would be shown to the user for confirmation. During this process, the user can override the recommendations. An interface of the modified *Package Installer* is shown at the right side of Fig. 2. Note that, once a sub-permission is checked in this interface, which means the user allows the access of the corresponding information or usage, *FMC* would not enforce its mapped policy.

After the user confirms or adjusts the permission configuration, if some of the sub-permissions are not granted, i.e., the three-bits is not equal to *000*, the Policy Enforcement Module in *FMC* will take the corresponding countermeasures to protect users' acoustic security and privacy. The overall deployment of *FMC*'s Policy Enforcement Module in Android 8.0 is shown in Fig. 3 as it runs through almost every level of Android, including the Java Native Interface (JNI) layer, which provides native interfaces for Android apps, and might help these apps bypass the current Android permission

1. Sequence Semantic Embedding. https://github.com/eBay/Sequence-Semantic-Embedding

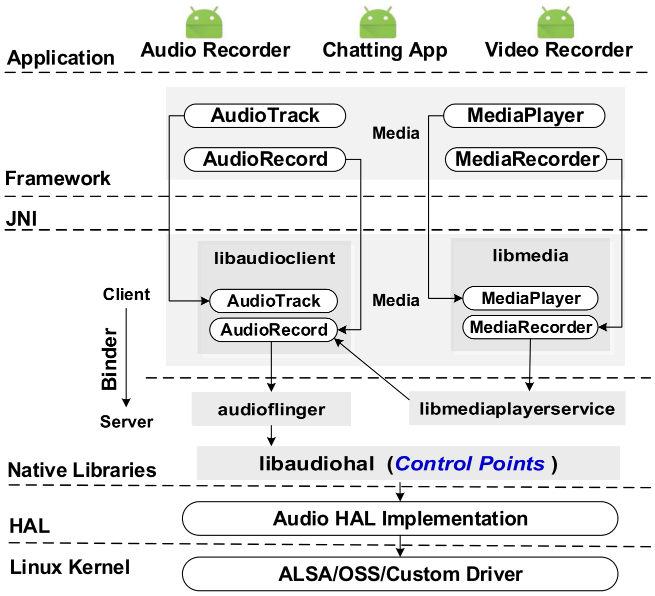


Fig. 3. The overall employment of *FMC* in Android 8.0. Three control points to enforce *treble policy*, *timbre policy*, and *exclusion policy* root in `streamHalHidl` lying in *libaudiohal* separately.

mechanism in the framework layer. Specifically, we modify the audio framework of Android system, and our main control points are rooted at the native layer to ensure that no bypassing attack at the Android framework layer is feasible.

To show how our Policy Enforcement Module works, we take `AudioRecord`, an Android recording API, as an example. At the application layer, after the sub-permissions of a certain app are configured as recommended or manually set by a user, a configuration item, which mainly consists of user identification (UID), package name and the three-bits permission list, is saved in a SQLite database. The database, named *fmc.db*, is created and managed by a system-level built-in app named as *fmcServer*. Note that, root privilege is required when trying to make modification to the system app of Android, thus our *fmcServer* and *fmc.db* can only be modified by system-level apps, such as *Package Installer*. Every time an app invokes the API of `AudioRecord`, we obtain its package name and UID at the native libraries layer of Android audio framework, and query *fmcServer* which provides a `ContentProvider` to fetch the corresponding sub-permissions configuration stored in *fmc.db*. The configuration is delivered within the audio framework with the help of Binder Inter-Process Communication (IPC), which follows the arrow flow as shown in Fig. 3. Finally, the app's configuration of the corresponding sub-permissions is sent to our control points, whose positions at `streamHalHidl` lie in *libaudiohal*.

Three control points to enforce the *treble policy*, *timbre policy* and *exclusion policy* work in `streamHalHidl` separately, although they take duties similarly, i.e., querying the configuration of the sub-permissions and enforcing the corresponding policies accordingly.

- For the *treble policy*, a low-pass filter, which is a six order Butterworth filter, is inserted to filter out information whose frequency is above 8 kHz (under the sample rate of 44.1 kHz). We use the Butterworth filter rather than an ideal filter to balance the time consumption and the low-pass filtering effect. Note that, we would do nothing but return the original audio data when the sampling rate of a recording task is set to smaller than 16 kHz, because, according to Nyquist sampling theorem, only information whose frequency is under half of the sample rate is preserved.
- For the *timbre policy*, we add a pitch shifter to interfere with the acoustic features of human voice. Namely, we change the pitch or disturb the *MFCC* of the recorded audio while maintaining its speed. We choose smbPitchShift [23], which is a classical and robust algorithm using Short-Time Fourier Transform (STFT), to complete pitch shifting.
- For the *exclusion policy*, we monitor the recording and playing states of the device in real time, then ensure that no audio playing happens when the microphone is in use. To achieve monitoring, we add a section of codes in `StreamHalHidl`, which keeps track of the exact current state of audio recording (from subclass `StreamInHalHidl`) and media playing (from subclass `StreamOutHalHidl`). As mentioned, the configuration parameters have already been sent to `StreamHalHidl`. Every time, when the audio data to be played is going to be written in the buffer, the added codes would check the configuration parameters. If exclusion control is required, once we find that the microphone and speaker are both in use, we replace the current audio fed to the speaker with silent content. When the recording ends, we stop the replacement at once and return the speaker to its original playing state.

Note that, all the monitoring and execution codes of *FMC* are running in processes different from the monitored third-party apps. Besides, they possess different UIDs. As a result, we effectively stop the monitored apps from bypassing or tampering with *FMC*'s monitor and processing.

6 *FMC*: EVALUATION

In order to measure the overhead brought by *FMC* and examine if *FMC* defends against acoustic attacks effectively while preserving usability, in this section, we conduct a series of evaluation experiments. All evaluation experiments were run on two Nexus 5X, which are shown in Fig. 1. The devices are equipped with 6 core CPU, 2 GB of RAM, and run Android 8.0.

6.1 Performance

Operational Latencies. It is necessary to measure the delays incurred by *FMC*, because latencies are crucial for acoustic services. In this part, we evaluate the audio playing and recording latencies brought by *FMC*.

We use the *audio latency* defined by Google[2] to measure the latencies of the acoustic components. We also use the audio latency increment to represent the overhead of our framework *FMC*. For audio apps, there are five common types of audio latencies, which are *Audio output latency*,

2. Audio latency: https://developer.android.com/ndk/guides/audio/audio-latency.html

TABLE 2
Audio Round-Trip Latency When Different Policies are Applied Respectively

Microphone source	Primary System	*FMC* with Low-Pass Filter in effect	*FMC* with Acoustic Feature Destruction in effect	*FMC* with Low-Pass Filter & Acoustic Feature Destruction in effect
VOICE_RECOGNITION	25.43	26.85	**35.96**	**38.75**
MIC	23.16	25.41	28.87	29.26
VOICE_COMMUNICATION	23.52	29.47	**35.80**	**37.73**
CAMCORDER	23.30	28.76	29.47	31.63
REMOTE_SUBMIX	15.55	16.53	20.29	20.34

All the units of data presented in the table are milliseconds (ms). An audio source defines both a default physical source of audio signal, and a recording configuration. Different audio sources will show different pre-processing latencies.

Audio input latency, *Round-trip latency*, *Touch latency*, and *Warmup latency*. We choose *Round-trip latency*, representing the sum of input latencies, app processing time and output latencies, as a major index to measure the operational latencies brought by the low-pass filter, which implements the *treble policy*, and acoustic feature destruction, which implements the *timbre policy*. A rough estimation of *Round-trip latency* is acquired by the testing app provided by Google. Specifically, Larsen test[3] is used to perform a *round-trip latency* test. We test each of the five microphone sources 20 times, then compare the latencies introduced with those of an unmodified system.

In all experiments conducted in this section, the sampling rate is set to 48 kHz; the player buffer and record buffer are both 192 frames when the audio thread type is native (JNI). That is, they are both 1,920 frames when the audio thread type is Java; the buffer test duration is 5 seconds; the number of simulated load threads is 4; and the mono channel setting is used.

As shown in Table 2, the acoustic feature destruction shows the longest latencies. When we set `VOICE_RECOGNITION` and `VOICE_COMMUNICATION` as microphone sources, the latencies with acoustic feature destruction triggered is about 50 percent more than that of the original system. However, as all the incurred latencies are within 40 milliseconds (ms), user experiences are barely affected.

We also conduct an experiment to evaluate the latencies, or reaction time, when *FMC* switchs between audio recording and audio playing. Since *FMC*'s *exclusion policy* prohibits the simultaneous usage of microphones and speaker, there is some reaction time for the framework to switch between playing and recording when the sub-permission of `ACOUSTICS_EXCLUSION` is not granted, which means that the *exclusion policy* is enforced. In order to measure the latencies, we play a clip of music in the background. At the same time, we open a recording app and start to record and then stop recording. We record the whole process and analyze the time differences by counting video frames in order to get a rough result of delays.

In general, the latencies brought by the switch-over are around 200$\sim$300 ms. The average reaction time of switching from recording to the playing state is 224 ms, while the average reaction time of switching from playing to the recording state is 297 ms, as recorded in 10 dependent tests we have conducted. These latencies are caused only when one function (audio recording or audio playing) acts while another function is forced to stop. The latencies seems a little high, but they do not happen during continuously playing or recording in, e.g., VoIP apps, which should not be enforced the *exclusion policy*. As a result, the user experiences in this experiment are barely affected by these latencies.

In fact, there is hardly any observable delay or abnormality during the entire testing process. We believe that the operational latencies brought by *FMC* are acceptable.

System Overhead. FMC runs across the application, application framework, and native libraries layers of Android. It might cause performance degradation and bring about system overhead. To quantify the system-wide overhead, we make a comparison of *AnTuTu* [24] (a popular benchmarking tool) scores with and without *FMC*. We benchmark Nexus 5X with and without *FMC* five times respectively, and the average results are shown in Table 3. We conclude from Table 3 that *FMC* only imposes a negligible overhead of 1.06 percent on the overall system.

6.2 Effectiveness of Balancing Security and Usability

We evaluate whether *FMC* can defend against the aforementioned acoustic attacks without affecting apps' normal functionalities. In addition, we demonstrate the effectiveness of *FMC*'s acoustic feature destruction, which can defend against the acoustic fingerprint interference attack.

6.2.1 *Functional Effects of* FMC *on Apps*

We choose a mainstream speech recognition app, *Otter Voice Meeting Notes*[4] (*Otter*), to evaluate whether apps can work as normal even when *FMC* is in effect.

Our evaluation scheme is as follows: First, we select 10 pieces of inaugural speech corpus. For each speech, we randomly select several 3 to 5 minutes sound clips, each containing 600 to 800 words and then let Google Text-to-Speech (TTS) read these texts in male and female tune separately. In the meantime, we use *Otter* to recognize the voice with or without *FMC* running. Under the setting that *FMC* is running, attempts are made for `ACOUSTICS_TREBLE` revoked only (i.e., the *treble policy* is enforced), `ACOUSTICS_TIMBRE` revoked only (i.e., the *timbre policy* is enforced), and both revoked (i.e., both policies are enforced). At last, we compare the recognition results with the original speech texts to check whether *FMC* affects *Otter*'s recognition accuracy. The text comparison tool that we choose is Tools 4 noobs.[5] Note that,

3. Larsen test. https://source.android.com/devices/audio/latency_measure.html#larsenTest

4. Otter Voice Meeting Notes. Available: https://play.google.com/store/apps/details?id=com.aisense.otter.

5. Tools 4 noobs. https://www.tools4noobs.com/online_tools/string_similarity/

TABLE 3
Average Results of *AnTuTu* Benchmarking Tests (The Integers Indicate the Benchmarked Points Given by *AnTuTu*, While the Numbers in Parentheses Indicate the Expected Range of Values With a Confidence Interval of 95%)

	with *FMC*	w/o *FMC*	overhead
CPU Mathematics	3,905 (86.36)	3,944 (14.36)	0.99%
CPU Common Use	3,395 (47.67)	3,409 (23.43)	0.39%
CPU Multi-Core	15,588 (1,142.94)	15,843 (1,304.32)	1.61%
GPU	22,941 (43.21)	22,935 (21.35)	−0.02%
UX Data Security	3,034 (8.46)	2,984 (4.90)	−1.66%
UX Data Processing	3,194 (11.56)	3,238 (39.03)	1.35%
UX Image Processing	3,732 (325.10)	3,996 (414.01)	6.61%
User Experience	7,811 (115.57)	7,802 (55.88)	−0.11%
RAM	1,845 (25.44)	1,838 (18.52)	−0.36%
ROM	2,522 (191.55)	2,709 (3.31)	6.90%
Overall	67,967 (1,181.51)	68,699 (1,714.75)	1.06%

we choose to use Google TTS to read the texts instead of reading them ourselves, because we try to eliminate individual variances caused by different speakers' accents, which may further interfere the accuracy of the evaluation.

The results show the deployment of *FMC* has no negative effect on the functionality of this popular app. No obvious drop in the recognition accuracy is witnessed with the *treble policy*, compared with the result we get under the primary system setting. Although a little fluctuation is observed with the *timbre policy*, the lowest recognition accuracy is still 94 percent, which is acceptable because the regular recognition accuracy is around 95 to 97 percent. Therefore, we consider that *FMC* preserves the usability of apps well. This is due to the frequencies of human voice that are mainly distributed in a low-frequency interval.

6.2.2 Effectiveness of Acoustic Fingerprint Interference

Since acoustic fingerprinting is one of the most severe threats which *FMC* faces, in this part, we conduct an experiment to evaluate the effectiveness of *FMC*'s acoustic feature destruction, namely, acoustic fingerprint interference.

FMC*Effectively Perturbs the Acoustic Feature Contained in Voice.* One of the most popular usage scenarios of the acoustic fingerprint is voice lock. Many *Finance* or *Social* apps utilize human voice as *fingerprint* to verify their users' identities before sensitive operations like confirming a payment or unlocking a device. Basically, acoustic features are extracted by these apps to construct a voice fingerprint during the initialization process. Ideally, *FMC* should render such functions ineffective when the `ACOUSTICS_TIMBRE` permission is not granted, no matter whether it is for payment confirmation or device unlocking.

We choose *Alipay*,[6] *WeChat*[7] and *Google Assistant*[8] to evaluate whether our *timbre policy* is strong enough to defend against such acoustic fingerprinting threats, among which *Alipay* and *WeChat* use the acoustic fingerprint as voice lock to do login, and *Google Assistant* to unlock a device.

6. Alipay. Available: https://play.google.com/store/apps/details?id=com.eg.android.AlipayGphone.
7. Wechat. Available: https://play.google.com/store/apps/details?id=com.tencent.mm.
8. Google Assistant. Available: https://play.google.com/store/apps/details?id=com.google.android.apps.googleassistant.

At first, we initialize these three apps without *FMC*, thus to make sure that the apps can capture the acoustic features in experimenters' voice exactly. During the training, for *Alipay* and *WeChat*, each of experimenters is asked to read out a number with eight digits while for *Google Assistant*, each of experimenters is required to say "OK, Google" and "Hey Google" twice respectively.

After that, we turn on *FMC* with `ACOUSTICS_TIMBRE` revoked. Ten login attempts are made to enter the apps of *Alipay* and *WeChat* through the same experimenter's voice, just as we set. Similarly, we let the same experimenters read out "Ok, Google" ten times to unlock the devices. If the acoustic feature destruction of *FMC* works well, the login or the unlock would fail. That is, the acoustic features of the experimenter could be totally disturbed by *FMC*. Then, the apps would not *recognize* the experimenter's voice.

- For *Alipay*, which works perfectly for voice login without *FMC*, fails to pass any test among all the ten tests where `ACOUSTICS_TIMBRE` is revoked.
- For *WeChat*, no login success when `ACOUSTICS_TIMBRE` is revoked.
- For *Google Assistant*, the results show that the device stays in lock state when `ACOUSTICS_TIMBRE` is revoked, just as expected.

The results indicate that the *timbre policy* provided by *FMC* radically decreases the recognition success rate of voiceprint authentication in *Alipay*, *WeChat* and *Google Assistant* to a great extent. They further show the effectiveness of *FMC*'s acoustic feature interference.

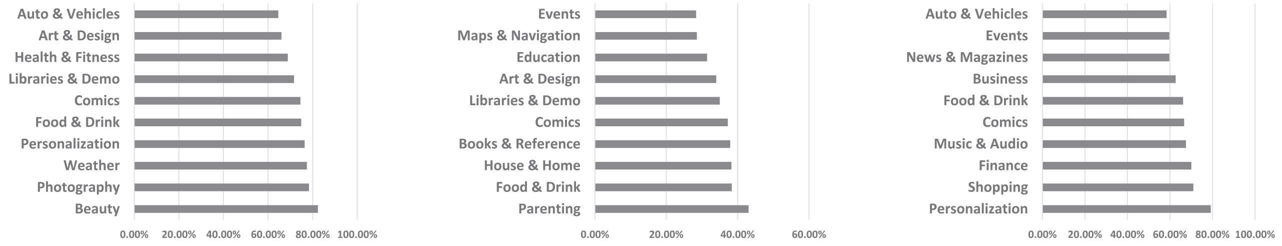
FMC*Perturbs the Generation of Acoustic Fingerprints With a Randomized Pattern.* Acoustic fingerprints generated with *FMC*'s acoustic fingerprint interference should be different every time. Otherwise, attackers could still make use of the fixed fingerprint to track the device.

In order to verify the effects that *FMC* perturbs the acoustic fingerprints differently every time, we conduct another evaluation experiment where we employ *Google Assistant*'s *unlock with voice match*. Different from the above experiments, the initialization process is done with *FMC*'s *timbre policy* enabled. Once the initialization is completed, we let the same experimenters try to unlock the devices. Note that, both the initialization and the unlock tests are performed while *FMC*'s *timbre policy* is enforced. If there is an obvious and fixed pattern of interference, the experimenters would unlock the devices successfully. However, in the experiment, no successful unlock occurred among all 20 attempts.

The result shows that *FMC*'s acoustic fingerprinting interference is robust and strong. Meanwhile, this experiment also verifies that even for acoustic fingerprints from the same person, they would present different features after *FMC*'s processing. This process prevents the attackers from guessing *FMC*'s interference pattern reversely.

6.3 Compatibility

We inspect *FMC*'s compatibility with 80 apps. We download the top 200 free apps from *Google Play Store*, and pick out all apps requiring `RECORD_AUDIO` permission. After that, we obtain 58 apps in total (29.0 percent). In addition, we randomly select 22 apps that do not require `RECORD_AUDIO` permission from the remaining apps. Thus, we get 80 apps.



(a) *treble policy* (b) *timbre policy* (c) *exclusion policy*

Fig. 4. Top categories on which the control policy should be enforced, i.e., the recommendation is to deny access. The x-axis presented the percent of apps in the category.

We run these apps manually one by one with *FMC* coming into effect, then observe whether these apps function as normal without crashing or errors. For those apps requiring `RECORD_AUDIO`, we apply different combinations of the *treble policy*, *timbre policy* and *exclusion policy* according to the functions of the apps and act as a user to trigger their functions.

The result of the above experiment shows that, when testing all popular 80 apps, no abnormality, neither crash nor error, happens during the compatibility test. Therefore, we argue that *FMC* has high compatibility.

6.4 Code Base

Here, code base[9] refers to a whole collection of source code that is used to build *FMC* in Android devices. It is a critical index to judge the framework's robustness and stability, since a heavy code base would lead to more uncertainty and increase maintenance costs. The modification that *FMC* made mainly concentrates on two parts: one is in the media module of Android 8.0, the other is in Android built-in app *Package Installer*. For the media module of Android, we modify 15 files among which there are 7 header files with only one or two lines of codes added. `StreamHalHidl.cpp`, lying in the native layer of Android, is the most heavily modified file, with 339 lines of code added or modified. For *Package Installer*, some minor modifications are made to `AndroidManifest.xml` and `PackageInstallerActivity.java`. Besides, we add four new files, namely `FmcContentProvider.java`, `FmcNewPermissionActivity.java`, `fmc_new_permission.xml` and `fmcServer.java`, to help message passing and demonstration. For this part, 52 lines of codes are modified and four files with 543 new lines are added. Besides, to store the sub-permission configurations of installed apps,*FMC* performs CURD (Create, Update, Retrieve, Delete) operations through APIs provided by `ContentProvider`, which is based on SQLite.

In summary, *FMC* brings a small code base with about a thousand lines of codes in Android devices, which is easy to test and maintain.

7 Measurement: An Empirical Study

In this section, we conduct an empirical study on microphone usages and microphone-related access control requirements on apps. The empirical study shows the protection coverage of *FMC*, as well as the application scopes of the three sub-permissions. These apps in the empirical study are crawled from a mainstream app market, *Google Play Store*. The labeled dataset we use is the same as shown in Table 1.

Among all 33,972 apps, 49.71, 20.94 and 52.35 percent apps are classified as suitable for enforcing the *treble policy*, *timbre policy* and *exclusion policy* respectively, which means that the corresponding sub-permissions should be deprived. These apps are all top in the app market. However, an obvious percentage gap exists between the *timbre policy* and the other two policies. This is due to the limited application scenario of the *timbre policy*.

Furthermore, we inspect each policy individually in Fig. 4. For the *treble policy*, nearly half of the apps from all 32 categories are recommended to enforce the policy. In Fig. 4a, *Beauty* apps, with an overall recommendation percentage of 82.35 percent, rank the top among all the categories, which may be attributable to the small sample size of *Beauty* apps. The second category is *Photography* and the third is *Weather*. Basically, *Beauty*, *Photography* and *Weather* mainly consist of specialized utility tools with limited microphone usage scenarios. It is reasonable to enforce a stricter microphone acoustic permission control on apps from these categories to avoid the abuse of private voice information by the apps.

Interestingly, *Parenting* category ranks at the top when we evaluate the *timbre policy* labels. The usage scenario of *Parenting* apps makes it easy for malicious apps to gather voice data. It is to say that*Parenting* apps usually require interactions with sound. Thus, these apps could easily defraud users and be granted the permission of using microphones. As a result, if the microphone permission is acquired by the apps of*Parenting*, it is better for these apps to apply a strict and finer permission control over the permission. Such apps can also come from *Food&Drinks*, *House&Home*, *Maps&Navigation*, and so on.

For the labels of the *exclusion policy*, *Personalization* and *Shopping* occupy the top two, followed by *Finance* and *Music & Audio*. Considering the usage scenarios of apps of *Music&Audio*, it is unusual for a *Music&Audio* app to use microphones and speakers at the same time.

Note that, some cases do exist as some interactive apps like Karaoke in *Entertainment* may require microphones and speakers working at the same time. For these apps, we would like to let users have more decision-making power according to their actual needs (perhaps the users like to use earphones instead when singing for better effects).

Further, we study the overall distribution of the three-bits recommendations. The results are shown in Fig. 5.

Considering the original category distribution of the dataset, we can see that only 25.74 percent of apps are tagged as

9. The code base of FMC is now at https://github.com/fduDaslabFMC/FMC/

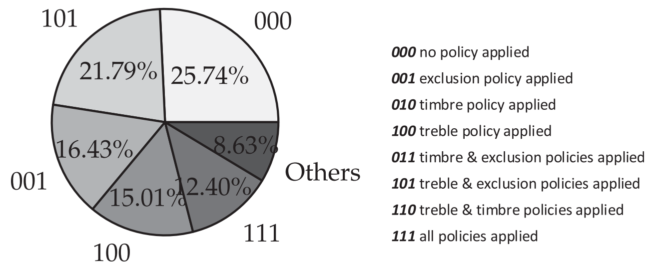


Fig. 5. Distribution of labeled three-bits recommendations.

needing all three permissions (*000*). While another 21.79 percent of the overall datasets, are tagged with *101*, standing for treble and exclusion restrictions are needed. *001* and *100* rank behind, holding 16.43 and 15.01 percent respectively.

Through the above empirical study, we conclude that the abuse of microphone permission exists, and a wide spectrum of apps should be restricted with finer audio permission controls.

8 Limitations and Future Work

Rationality of Permission and Policy Design. In *FMC*, we design and define three fine-grained permissions, i.e., `ACOUSTICS_TREBLE`, `ACOUSTICS_TIMBRE` and `ACOUSTICS_EXCLUSION`, derived naturally from the acoustic features and categories of existing attacks. However, the rationality of the permission design needs to be further studied, according to the advancement of threats on microphones in mobile devices.

For now, the proposed *FMC* does not focus on context, but it is highly flexible and can be extended to incorporate other context-aware solutions [25], [26], [27], [28], where the audio data are controlled as a whole.

Dataset Assumptions. The dataset we are using is pulled and maintained manually, and we assume that for a given app to be installed, its corresponding description is given in advance. It means that *FMC* cannot make a prediction once the app to be installed is out of the stored dataset or the app is from another app market. To make up for the deficiency, we plan to create an interaction process between our server and several third-party app markets to dynamically query and pull the descriptions according to the apps' package names. Alternatively, we can also further develop an extended version specifically for those apps that cannot be found in app markets and for those with non-English descriptions. An advanced prediction model can be built based on a training set of more dimensions (e.g., package name, static code scanning reports) in addition to descriptions and the declaration of `RECORD_AUDIO` permission. In addition, the scale of our dataset used to train the prediction model in *ADTP* can be further extended, since there is still room for improvement in the prediction accuracy. We plan to conduct experiments with a more diverse set of NLP tools to develop a model with higher performance. Our fundamental goal is to realize a more intuitive prediction with high accuracy.

Model Training Accuracy and Dynamic Policy Update. In *FMC*, we train an NLP model, named *ADTP*, to complete the description-to-permission translation. As described in Section 5.1, the prediction accuracy is 82.82 percent, which means that about 17 percent of the recommendations are inaccurate. Under this circumstance, the only remedy that can be made is the manual check by users. The limited accuracy would weaken the entire solution by relying on users as the last line of defense. However, it would not introduce new privacy leakage compared with the current all-or-nothing model, as it simply provides the options to impose more restrictions on existing access level.

The current *FMC* framework finishes its policy decision at installation time. Once the policies are put into effect, we cannot update or remove the permission choices and policy implementation during runtime dynamically. In the future, a function to re-adjust policies at runtime will be implemented.

Attack Exceptions. For those attacks completed at the physical level, *FMC* only plays a limited role because it is a software based permission framework. For example, *DolphinAttack* [15] utilizes the non-linearity characteristic of microphone circuits in mobile devices, where voice commands are modulated on ultrasonic carriers and are further demodulated by circuits to normal low frequency signals that can be directly recognized by speech recognition software. The down-mixing happens before the audio data are converted to digital signals. Then the digital logic, such as *FMC*'s *treble policy*, would not treat it as high frequency component. Therefore, *FMC* would not restrict its access. That is, the app is authorized at the software level, thus could be comprised under *DolphinAttack*.

We do not owe this to *FMC*'s limited capability because *FMC* only targets adversaries residing in the victim's phone and restricts their behaviors with a finer permission control. However, *DolphinAttack* is a result of ultrasound played by a remote device, to which end *FMC* needs not and could not restrict. Note that, Zhang *et al.*, the discoverers of *DolphinAttack*, also proposed several practical countermeasures against *DolphinAttack* from both the hardware level and software level. These countermeasures are compatible with our *FMC*, thus can be integrated with *FMC*.

9 Related Work

Attacks leveraging sensors, like microphones, in smart devices are a hot topic in the field of Android security. Researchers provide some common or specific solutions to defend against these attacks.

Attacks via Sensors in Mobile Devices. Sensor-based attacks in mobile devices have caused widespread concerns in recent years. There were a number of works trying to launch such attacks.

Among all the different kinds of sensors, motion sensors (e.g., accelerometers, gyroscopes), were popular because of their zero-permission characteristics in mobile devices. Cai *et al.* demonstrated an attack which could extract features from devices' orientation data to infer keystrokes, and developed *TouchLogger* [29] as a prototype. Other similar works included *TapLogger* [30], which could infer user inputs on touchscreens. Dey *et al.* [31] showed that the accelerometer was also a key source of side channel attacks to perform privacy inference or device tracking. Yan *et al.* [32] proposed *Gyrophone* which showed that gyroscope data from smartphones were enough to identify speaker information.

Besides, the attacks based on media sensors (i.e., camera and microphone) also emerged. And such attacks could sometimes be accomplished by multiple sensors. *PlaceRaider* [33], through the combined use of camera and other sensors, built three dimensional models of indoor environment then stole virtual objects. Simon *et al.* [6] used microphone and camera, correlating with the layout of digits on smartphones, to realize PIN inference on smartphones. There were also fingerprinting works using microphones [3], [4]. The basic idea was to uniquely identify an individual or device through playing and recording audio samples, and analyze the extracted sound features. Another example named *SoundComber* [5], which designed a trojan app with a few innocuous permissions, to extract private information from audio sensors in a phone. Moreover, Bojinov *et al.* [16] implemented an accelerometer-based fingerprinting and a speakerphone-microphone fingerprinting. Last but not least, *DolphinAttack* [15] successfully hit speech recognition systems, such as *Siri* and *Google Now*, through inaudible voice commands injections, using the perceptual differences between acoustic components and human ears.

Defending Against the Attacks on Sensors. In recent years, researchers also proposed several solutions to defend against the attacks mentioned above. Machiraju *et al.* [34] explored the vulnerabilities of mobile phones against sensor-sniffing attacks, further proposed a general framework, and discussed various possible approaches to fully implement the framework. Das *et al.* [35], [36] discussed the defenses and countermeasures against fingerprinting through mobile devices via large-scale user studies. Beresford *et al.* [37] proposed *MockDroid*, which was a modified version of Android OS, allowing users to provide fake or *mock* data, including sensor data, to apps. Han *et al.* [38] proposed *senDroid*, which audited GPS, camera, microphone and standard sensor access in Android by hooking, and provided auditing reports to users. Different from the above works, which provided general countermeasures or frameworks trying to manage several different kinds of sensors at the same time, *FMC* focuses on the microphone management only.

Petracca *et al.* [39] proposed *AuDroid*, which tracked the creation of audio communication channels explicitly and controlled the information flow over these channels to prevent several types of voice control attacks in mobile devices. However, it differs from *FMC* at the implementation layer and attack scenarios.

In addition, *FMC* is the first work to finely control the audio data and acoustic components according to their physical and concurrent features.

Fine-grained Permission Control in Mobile Devices. The coarse permission control of Android had been under discussion for years. There were several works trying to improve the existing permission control framework. *FlaskDroid* [40] provided a generic security architecture for the Android OS. *FlaskDroid* could serve as a flexible and effective ecosystem to instantiate different security solutions. It defended against permission-related attacks from third-party applications at Android framework layer. Rashidi *et al.* [41] proposed a crowdsourcing recommendation framework, implemented as *RecDroid*, that facilitated a user-help-user environment when controlling smartphone permissions. Different from *FMC*, *RecDroid* did not create new fine-grained permissions, and the feature of *fine-grained* here means it controlled permission at a system service level. *PolEnA* [42], which was an extension of Android Security Framework (ASF), allowed for the definition of fine-grained security policies and their dynamic verification. Rui *et al.* [43] proposed a usage and access control model, and provided a permission-based mandatory access control at Android framework layer, Linux kernel, and hardware layer. It avoided permission leakages via the ARM TrustZone security extension mechanism.

Smart and Context-Aware Permission Solution in Mobile Devices. To address the rigidity of OS permission administrations and their mismatch with users' privacy preferences, machine learning technologies were quite often used by prior works to assist users in deciding their permission strategies in mobile devices. Bilogrevic *et al.* [44] proposed *SPISM*, which adapted to each user's behavior, and predicted the level of detail for each sharing decision without revealing any private information. Qu *et al.* [45] proposed *Autocog* which leveraged NLP technologies to verify or configure permissions of Android apps. Recently, Gasparis *et al.* [46] proposed *Figment* which provided a set of libraries to enforce dynamic and contextual access control for Android apps. Chen *et al.* [47] leveraged the technologies of NLP to identify hidden privacy settings in mobile apps. *FMC* extends the prior works of the machine learning based policy recommendation to implement an NLP-based policy prediction and recommendation module, referred as to *ADTP*, for three new-defined sub-permissions in *FMC*. Here, *FMC* only extends the standard permission model of Android, thus does not employ the feature of the context-aware access control mechanism. This feature does not conflict with three new policies proposed in *FMC*. They can cooperate with each other in a combined access control model. But it is not a contribution in this paper.

10 Conclusion

In this paper, we propose *FMC* which provides a fine-grained and smart access control framework over microphones on Android, offering users more granular control over their audio data and ability to restrict the concurrent usage with other acoustic components. By adding three finer permissions, `ACOUSTICS_TREBLE`, `ACOUSTICS_TIMBRE` and `ACOUSTICS_EXCLUSION`, the existing Android permission control framework is enhanced in security. In addition, *FMC* leverages NLP technologies to intelligently recommend permission settings to users. Evaluations in this paper show that *FMC* protects users from microphone-based attacks with an acceptable performance overhead of 1.06 percent, and *FMC* is promising in terms of its compatibility and effectiveness. In addition, to the best of our knowledge, this paper is the first research work to explore the policy of DSoD at the operating system level.

Acknowledgments

This work was supported by NSFC (Grant No.: 61572136, U1836207), the National Key R&D Program of China (Grant No.: 2018YFC0830900), and STCSM (Grant No.: 18511103600). The authors would like to thank anonymous reviewers for their insightful comments. The authors would also like to thank Dr. Kai Zhang and Dr. Xinyi Zhang for their English polish.

REFERENCES

[1] P. R. Center, "Google play store apps permissions," 2018, Accessed: Aug. 28, 2019. [Online]. Available: https://www.pewinternet.org/interactives/apps-permissions/

[2] I. Amerini, R. Becarelli, R. Caldelli, A. Melani, and M. Niccolai, "Smartphone fingerprinting combining features of on-board sensors," *IEEE Trans. Inf. Forensics Secur.*, vol. 12, no. 10, pp. 2457–2466, Oct. 2017.

[3] A. Das, N. Borisov, and M. Caesar, "Do you hear what I hear?: Fingerprinting smart devices through embedded acoustic components," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 441–452.

[4] Z. Zhou, W. Diao, X. Liu, and K. Zhang, "Acoustic fingerprinting revisited: Generate stable device ID stealthily with inaudible sound," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 429–440.

[5] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang, "Soundcomber: A stealthy and context-aware sound trojan for smartphones," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2011, pp. 17–33.

[6] L. Simon and R. J. Anderson, "PIN skimmer: Inferring pins through the camera and microphone," in *Proc. ACM Workshop Secur. Privacy Smartphones Mobile Devices*, 2013, pp. 67–78.

[7] 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 *Proc. 22nd Annu. Int. Conf. Mobile Comput. Netw.*, 2016, pp. 29–41.

[8] N. Roy, H. Hassanieh, and R. R. Choudhury, "BackDoor: Making microphones hear inaudible sounds," in *Proc. 15th Annu. Int. Conf. Mobile Syst. Appl. Services*, 2017, pp. 2–14.

[9] L. Deshotels, "Inaudible sound as a covert channel in mobile devices," in *Proc. 8th USENIX Workshop Offensive Technol.*, 2014, p. 16.

[10] Z. Xu and S. Zhu, "SemaDroid: A privacy-aware sensor management framework for smartphones," in *Proc. 5th ACM Conf. Data Appl. Secur. Privacy*, 2015, pp. 61–72.

[11] X. Wang, K. Sun, Y. Wang, and J. Jing, "Deepdroid: Dynamically enforcing enterprise policy on android devices," in *Proc. 22nd Annu. Netw. Distrib. Syst. Secur. Symp.*, 2015, p. 10.

[12] K. Olejnik, I. Dacosta, J. S. Machado, K. Huguenin, M. E. Khan, and J. Hubaux, "SmarPer: Context-aware and automatic runtime-permissions for mobile devices," in *Proc. IEEE Symp. Secur. Privacy*, 2017, pp. 1058–1076.

[13] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. A. Wagner, and K. Beznosov, "Android permissions remystified: A field study on contextual integrity," in *Proc. 24th USENIX Secur. Symp.*, 2015, pp. 499–514.

[14] S. A. Anand and N. Saxena, "A sound for a sound: Mitigating acoustic side channel attacks on password keystrokes with active sounds," in *Proc. 20th Int. Conf. Financial Cryptogr. Data Secur.*, 2016, pp. 346–364.

[15] G. Zhang, C. Yan, X. Ji, T. Zhang, T. Zhang, and W. Xu, "DolphinAttack: Inaudible voice commands," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 103–117.

[16] H. Bojinov, Y. Michalevsky, G. Nakibly, and D. Boneh, "Mobile device identification via sensor fingerprinting," *CoRR*, 2014. [Online]. Available: http://arxiv.org/abs/1408.1416

[17] Z. Fang *et al.*, "revDroid: Code analysis of the side effects after dynamic permission revocation of Android apps," in *Proc. 11th ACM Asia Conf. Comput. Commun. Secur.*, 2016, pp. 747–758.

[18] Z. Fang, W. Han, and Y. Li, "Permission based Android security: Issues and countermeasures," *Comput. Secur.*, vol. 43, pp. 205–218, 2014.

[19] W. Diao, X. Liu, Z. Zhou, and K. Zhang, "Your voice assistant is mine: How to abuse speakers to steal information and control your phone," in *Proc. 4th ACM Workshop Secur. Privacy Smartphones Mobile Devices*, 2014, pp. 63–74.

[20] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[21] N. Li, M. V. Tripunitara, and Z. Bizri, "On mutually exclusive roles and separation-of-duty," *ACM Trans. Inf. Syst. Secur.*, vol. 10, no. 2, 2007, Art. no. 5.

[22] R. A. Botha and J. H. P. Eloff, "Separation of duties for access control enforcement in workflow environments," *IBM Syst. J.*, vol. 40, no. 3, pp. 666–682, 2001.

[23] S. M. Bernsee, "Time stretching and pitch shifting of audio signals an overview," *DSP Dimension*, 2003. [Online]. Available: http://my.fit.edu/~vkepuska/ece3551/The%20DSP%20Dimension/Tutorials/TimeStretchingOverview.pdf

[24] AnTuTu, "Antutu benchmark 3D – Android apps on Google play," 2018, Accessed: Apr. 20, 2018. [Online]. Available: https://play.google.com/store/apps/details?id=com.antutu.benchmark.full

[25] E. Bertino, B. Catania, M. L. Damiani, and P. Perlasca, "GEO-RBAC: A spatially aware RBAC," in *Proc. 10th ACM Symp. Access Control Models Technol.*, 2005, pp. 29–37.

[26] J. Joshi, E. Bertino, U. Latif, and A. Ghafoor, "A generalized temporal role-based access control model," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 1, pp. 4–23, Jan. 2005.

[27] Y. Kim, C. Moon, D. Jeong, J. Lee, C. Song, and D. Baik, "Context-aware access control mechanism for ubiquitous applications," in *Proc. 3rd Int. Atlantic Web Intell. Conf.*, 2005, pp. 236–242.

[28] B. Shebaro, O. Oluwatimi, and E. Bertino, "Context-based access control systems for mobile devices," *IEEE Trans. Dependable Secure Comput.*, vol. 12, no. 2, pp. 150–163, Mar./Apr. 2015.

[29] L. Cai and H. Chen, "TouchLogger: Inferring keystrokes on touch screen from smartphone motion," in *Proc. 6th USENIX Workshop Hot Top. Secur.*, 2011, p. 9.

[30] Z. Xu, K. Bai, and S. Zhu, "TapLogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors," in *Proc. 5th ACM Conf. Secur. Privacy Wireless Mobile Netw.*, 2012, pp. 113–124.

[31] S. Dey, N. Roy, W. Xu, R. R. Choudhury, and S. Nelakuditi, "AccelPrint: Imperfections of accelerometers make smartphones trackable," in *Proc. 21st Annu. Netw. Distrib. Syst. Secur. Symp.*, 2014, p. 12.

[32] Y. Michalevsky, D. Boneh, and G. Nakibly, "Gyrophone: Recognizing speech from gyroscope signals," in *Proc. 23rd USENIX Secur. Symp.*, 2014, pp. 1053–1067.

[33] R. Templeman, Z. Rahman, D. J. Crandall, and A. Kapadia, "PlaceRaider: Virtual theft in physical spaces with smartphones," in *Proc. 20th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2013, p. 4.

[34] L. Cai, S. Machiraju, and H. Chen, "Defending against sensor-sniffing attacks on mobile phones," in *Proc. 1st ACM SIGCOMM Workshop Netw. Syst. Appl. Mobile Handhelds*, 2009, pp. 31–36.

[35] A. Das, N. Borisov, and M. Caesar, "Tracking mobile web users through motion sensors: Attacks and defenses," in *Proc. 23rd Annu. Netw. Distrib. Syst. Secur. Symp.*, 2016, p. 20.

[36] A. Das, N. Borisov, and E. Chou, "Every move you make: Exploring practical issues in smartphone motion sensor fingerprinting and countermeasures," in *Proc. Privacy Enhancing Technol.*, 2018, pp. 88–108.

[37] A. R. Beresford, A. C. Rice, N. Skehin, and R. Sohan, "MockDroid: Trading privacy for application functionality on smartphones," in *Proc. 12th Workshop Mobile Comput. Syst. Appl.*, 2011, pp. 49–54.

[38] W. Han *et al.*, "senDroid: Auditing sensor access in Android system-wide," *IEEE Trans. Dependable Secure Comput.*, 2017. to be published. doi: 10.1109/TDSC.2017.2768536.

[39] G. Petracca, Y. Sun, T. Jaeger, and A. Atamli, "AuDroid: Preventing attacks on audio channels in mobile devices," in *Proc. 31st Annu. Comput. Secur. Appl. Conf.*, 2015, pp. 181–190.

[40] S. Bugiel, S. Heuser, and A. Sadeghi, "Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies," in *Proc. 22th USENIX Secur. Symp.*, 2013, pp. 131–146.

[41] B. Rashidi, C. J. Fung, and T. Vu, "Android fine-grained permission control system with real-time expert recommendations," *Pervasive Mobile Comput.*, vol. 32, pp. 62–77, 2016.

[42] G. Costa, F. Sinigaglia, and R. Carbone, "PolEnA: Enforcing fine-grained permission policies in Android," in *Proc. Comput. Saf. Rel. Secur.*, 2017, pp. 407–414.

[43] R. Chang *et al.*, "Towards a multilayered permission-based access control for extending android security," *Concurrency Comput., Practice Experience*, vol. 30, no. 5, 2018, Art. no. e4180.

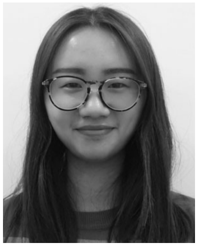
[44] I. Bilogrevic, K. Huguenin, B. Agir, M. Jadliwala, and J. Hubaux, "Adaptive information-sharing for privacy-aware mobile social networks," in *Proc. ACM Int. Joint Conf. Pervasive Ubiquitous Comput.*, 2013, pp. 657–666.

[45] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in Android applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 1354–1365.

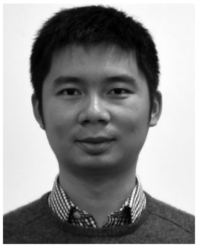
[46] I. Gasparis, Z. Qian, C. Song, S. V. Krishnamurthy, R. Gupta, and P. Yu, "Figment: Fine-grained permission management for mobile apps," in *Proc. IEEE Conf. Comput. Commun.*, 2019, pp. 1405–1413.

[47] Y. Chen *et al.*, "Demystifying hidden privacy settings in mobile apps," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 570–586.

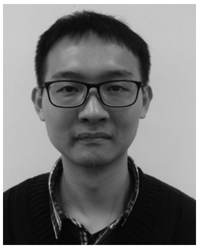
Weili Han received the PhD degree from Zhejiang University, Zhejiang, China, in 2003. Then, he joined the faculty of Software School, Fudan University, Shanghai, China. He is currently a full professor at Fudan University, Shanghai, China. His research interests are mainly in the fields of data systems security, access control, and digital identity management. He is a member of ACM, SIGSAC, IEEE, and a distinguished member of CCF. From 2008 to 2009, he visited Purdue University as a visiting professor funded by China Scholarship Council and Purdue University. He serves in several leading conferences and journals as PC members, reviewers, and an associate editor. He is a member of the IEEE.



Chang Cao received the BS degree from Fudan University, Shanghai, China, in 2016. She is working toward the graduate degree majored in computer software and theory at Fudan University, Shanghai, China. She is currently a member of the Laboratory for Data Analytics and Secur.. Her research interest mainly includes sensor-related access control in mobile devices.



Zhe Zhou received the PhD degree from the IE Department, The Chinese University of Hong Kong, Hong Kong, in 2017. He is currently a pre-tenure associate professor with the School of Computer Science, Fudan University, Shanghai, China. His research interests mainly include systems security and privacy protection.



Shize Chen received the BS degree from Shanghai University, Shanghai, China, in 2017. He is working toward the graduate degree majored in software engineering at Fudan University, Shanghai, China. He is currently a member of the Laboratory for Data Analytics and Secur.. His research interests include data analysis and blockchain.

Lingqi Huang is working toward the undergraduate degree majored in software engineering at Fudan University, Shanghai, China. His research interest include password security and systems security.

X. Sean Wang received the PhD degree in computer science from the University of Southern California, Los Angeles, California, in 1992. He is currently a distingished professor at the School of Computer Science, Fudan University, Shanghai, China. Before joining Fudan University in 2011, he was the dorothean chair professor in computer science at the University of Vermont, Burlington, Vermont. His research interests include data systems and data security. He is a member of the ACM and senior member of the IEEE, and a distinguished member of CCF.