Regulating and Securing the Interfaces Across Mobile Apps, OS and Users

by

Huan Feng

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2016

To my family and many other important people in my life.

ACKNOWLEDGEMENTS

I am glad to spend some of my best years in Real-Time Computing Laboratory, and would like to give special thanks to my advisor, Professor Kang Shin. During the five years of my Ph.D., I learned how to propose, formalize, design, implement, evaluate and sell an idea, which will be impossible without his guidance, support and patience. I especially appreciate the degree of freedom we have in the lab and his encouragement to pursue "something really excites you and makes you happy." These values endure beyond graduate school and research and are particularly rare in today's fast-paced society. I would also like to thank Professor Alex Halderman, Atul Prakash and Zhengya Zhang for serving in my dissertation committee and providing invaluable feedback.

I appreciate the help and support from the past and current members of the RTCL family. Kassem Fawaz has been a great friend and I will always remember the days and nights we work together and many of our long and enjoyable talks. I would also like to thank Dongyao Chen, Seunghyun Choi, Arun Ganesan, Sihui Han, Xiaoen Ju, Yu-Chih Tung and Michael Zhang for the useful discussions and feedback, and those good memories we shared together.

I would like to thank my family for their unconditional support through this journey to which I am deeply indebted. I am also grateful to the accompany of my friends and roommates, Kenneth Cheng, Yan Dong, Jing Zhang for their understanding and support during the past few years. The love of yours gives me courage and endurance to persist.

Finally, I would like to gratefully acknowledge support of the US National Science Foundation under Grants CNS-1114837 and CNS-1505785.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Regulating and Securing the Interfaces Across Mobile Apps, OS and Users

by

Huan Feng

Chair: Kang G. Shin

Over the past decade, we have seen a swift move towards a mobile-centered world. This thriving mobile ecosystem builds upon the interplay of three important parties: the mobile user, OS, and app. These parties interact via designated interfaces many of which are newly invented for, or introduced to the mobile platform. Nevertheless, as these new ways of interactions arise in the mobile ecosystem, what is enabled by these communication interfaces often violates the expectations of the communicating parties. This makes the foundation of the mobile ecosystem untrustworthy, causing significant security and privacy hazards. This dissertation aims to fill this gap by: 1) securing the conversations between trusted parties, 2) regulating the interactions between partially trusted parties, and 3) protecting the communications between untrusted parties.

We first deal with the case of mobile OS and app, and analyze the Inter-Process Communication (IPC) protocol (Android Binder in particular) between these two untrusted parties. We found that the Android OS is frequently making unrealistic assumptions on the validity (sanity) of transactions from apps, thus creating significant security hazards. We analyzed the root cause of this emerging attack surface and protected this interface by developing an effective, precautionary testing framework and a runtime diagnostic tool. Then, we study the deficiency of how a mobile user interacts with an app that he can only partially trust. In the current mobile ecosystem, information about the same user in different apps can be easily shared and aggregated, which clearly violates the conditional trust mobile user has on each app. This issue is addressed by providing two complementary options: an *OS-level extension* that allows the user to track and control, during runtime, the potential flow of his information across apps; and a *user-level solution* that allows the users to maintain multiple isolated profiles for each app. Finally, we elaborate on how to secure the voice interaction channel between two trusted parties, mobile user and OS. The open nature of the voice channel makes applications that depend on voice interactions, such as voice assistants, difficult to secure and exposed to various attacks. We solve this problem by proposing the first system, called *VAuth*, that provides continuous and usable authentication for voice commands, designed as a wearable security token. It collects the body-surface vibrations of a user via an accelerometer and continuously matches them to the voice commands received by the voice assistant. This way, VAuth guarantees that the voice assistant executes only the commands that originate from the voice of the owner.

This thesis gives a detailed examination of the privacy and security issues across various interfaces in the mobile ecosystem, analyzes the trust relationship between different parties and proposes practical solutions. We share the experience learned from tackling these problems, and hope it will be reused when dealing with similar issues in other domains.

CHAPTER I

Introduction

During the past decade, we have witnessed a swift move towards a mobile-centered world. In the U.S, the majority of all digital media consumption now takes place in the mobile ecosystem [67], and a typical smartphone user spends more than 3.5 hours per day on its smartphone device. Most of the time (more than 80 percent) is consumed purely on mobile apps [50] and this mobile experience is taking over every aspect of our digital life. For a large percentage of population in developing countries, a smartphone is their first and only access to the Internet, and becomes the daily vehicle of their payments, health, and even political systems.

Compared to traditional computing devices, such as desktops and laptops, a smartphone has several unique properties which make its security and privacy more critical. First, the smartphone is always connected to the Internet and always possessed by the user, which enables continuous user tracking and profiling. Second, the smartphone is rarely shared among different users, making it the very hub for personal information and activities. Third, the smartphone is equipped with various sensors, and can thus see/feel the user's physical environment. All of these make the smartphone an attractive target of attacks.

The mobile privacy and security issues are further aggravated by the prosperity and controversy of the mobile app ecosystem. As of the end of 2016, there are more than 2 million apps available for download in Apple's App Store and Google Play, respectively. Anyone can also become a developer of these apps, with a deployment fee of as low as $25, and deploy their apps at a global scale. These developers can be inexperienced, careless or even malicious, posing great threats to smartphone users and their information. Moreover, in some countries or areas of the world, there is no centralized app store and users can only download apps from untrusted sources and have to deal with hundreds of third-party app stores.

All of these challenges and threats make trust a complicated issue in the current mobile ecosystem. Different parties need to interact in a way that both fulfills the functional needs of each other and at the same time be aware of the potential hostility between themselves and from others. Nevertheless, we find that as more communication and interaction interfaces are introduced to the mobile ecosystem, what is enabled by these interfaces often fails to comply with the trust relationship between the two communicating parties. This makes the foundation of the mobile ecosystem untrustworthy and results in significant security and privacy hazards. In this chapter, we present an overview of the mobile ecosystem, analyze the trust relationship between different parties, describe the urgent problems to be addressed and summarize our approaches to resolving these issues.

1.1 An Overview of the Mobile Ecosystem

The mobile ecosystem centers around three parties: mobile user, OS, and app (see Fig. 1.1). The OS provides a set of API abstractions that app developers can directly work with. These APIs drive the hardware sensors, system UIs and networking and storage interfaces. This way, the app developer can focus on implementing the logic of the apps, instead of being burdened with system/hardware details. User installs apps on the smartphone OS, either manually, or through some centralized or third-party app store. These app stores serve as the delivery channel for mobile apps and the first

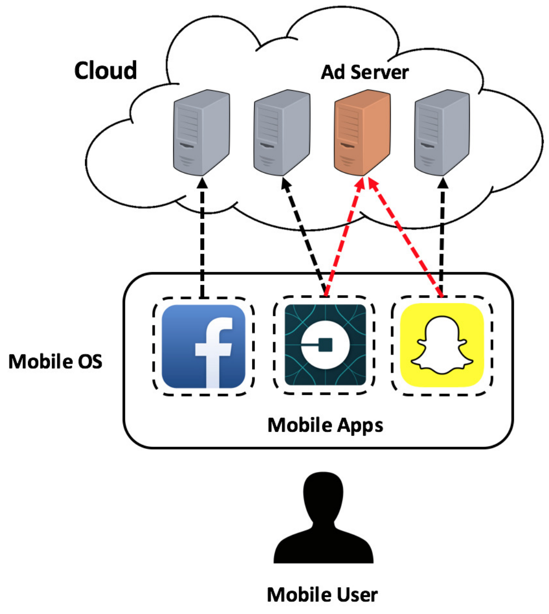


Figure 1.1: An overview of the mobile ecosystem.

defense line against malicious app contents. When the user interacts with an app, his usage behavior and other personal information will be collected, sent to the cloud, and then analyzed. In some cases, this is legitimate because most apps are merely UI abstractions and local content cache, while the actual functional logic resides in the cloud. Examples of these apps cover the categories of shopping, news, and social networks. However, in many cases, this tracking process is purely for advertising purposes. The major monetization channel for mobile apps is by incorporating third-party ad libraries which collect user demographics, locations, and behaviors, and enable more targeted advertising.

The dynamics of the mobile ecosystem becomes interesting when considering the trust relationship between these parties:

- **OS does not trust apps.** Apps running in an OS can be buggy or malicious, and hence a modern smartphone OS typically enforces a sandbox environment for the app's execution. Apps can only interface with APIs exported by the OS and extensive sanity checks are deployed around these APIs. In both An-

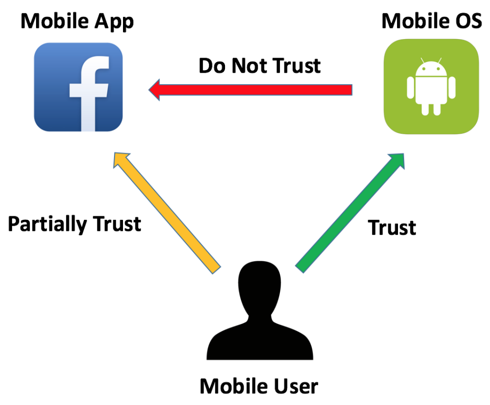


Figure 1.2: The trust relationship between three parties in the mobile ecosystem.

droid and iOS, sensitive APIs are guarded by a permission model. Each app needs to explicitly declare the permissions it requires to fulfill its functionalities. Specifically, Apple's iOS adopts a runtime permission model where the user will be prompted the first time when app tries to access certain resources, while Android adopted an install-time permission model, which requires all requested permissions to be reviewed during install-time. The runtime permission model is generally believed to be more effective than install-time enforcements, and the latest Android OS (6.0) has already made the switch to the former.

- **User partially trusts apps.** The user has to trust the app to some extent if (s)he needs the service provided by the app. It is impossible for the app to provide any useful service unless it acquires the necessary information. For example, if the user wants to purchase some products via a shopping app, information about the purchased items is unavoidably shared with this app. Nevertheless, the trust the user has on an app is not unconditional — he only trusts it with the information related to (generated in) the app. The user may trust a map app with his location traces, but will not trust it with his financial information or contact history.

- **User trusts OS.** The user needs to have a complete trust of the mobile OS.

In fact, if the OS is mal-intented (for example, in the case of a user installing malicious third-party ROM), any information stored or generated in the smartphone is inherently insecure. The user mostly interacts with OS through a touch screen or physical keyboard, but also starts to embrace a voice interaction channel. This is largely due to the increasing popularity of mobile voice assistant, such as Siri and Google Now. It is important to ensure the integrity of these interaction surfaces is not compromised, making sure the trust relationship between mobile OS and the user is not exploited.

1.2 Problems

This dissertation investigates the cross-party interfaces in the current mobile ecosystem. Many of these interfaces are either newly invented for, or introduced to the mobile world and does not fit the complicated trust relationship across parties. This dissertation aims to identify these gaps and 1) secure the conversations between trusted parties, 2) regulate the interactions between partially trusted parties, and 3) protect the communications between untrusted parties. Specifically, it focuses on the following problems.

1.2.1 Insecure Client-side IPCs

First, we deal with the case of two opposing parties, mobile OS and app, and analyze the Inter-Process Communication (IPC) protocol between them. In Android, communications between apps and system services are supported by a transaction-based IPC mechanism. `Binder`, as the cornerstone of this IPC mechanism, separates two communicating parties as client and server. As with any client–server model, the server should not make any assumption on the validity (sanity) of client-side transactions. To our surprise, this principle is found to have frequently been overlooked in the implementation of Android system services. We want to find why developers keep making this seemingly simple mistake and how to defend against emerging vulnerabilities on this attack surface.

1.2.2 Unregulated Cloud-side Profiling

Then, we study the deficiency of existing mobile user interactions with apps, the party that he can only partially trust. From the user's perspective, his behaviors in different smartphone apps capture different views of his life, and are largely independent of each other. These views are supposed to be kept as 'isolated islands' of information and stored separately. However, in the current mobile app ecosystem, a curious party can covertly link and aggregate usage behaviors of the same user across different apps. In fact, once the users' behavioral information is at the apps' hands, it may be shared or leaked in an arbitrary way without the users' control or consent. We refer to this as *unregulated aggregation* of app-usage behaviors. How to characterize and further reduce this threat remains a question to be answered.

1.2.3 Unprotected Voice Interaction Channel

Last, we turn our attention to the voice interaction channel between two trusted parties, mobile user and OS. Voice has become an increasingly popular User Interaction (UI) channel, largely contributing to the ongoing trend of wearables, smart vehicles, and home automation systems. Voice assistants such as Siri, Google Now, and Cortana, have become our everyday fixtures, especially in scenarios where touch interfaces are inconvenient or even dangerous to use, such as driving or exercising. Nevertheless, the open nature of the voice channel makes voice assistants difficult to secure and exposed to various attacks as demonstrated by security researchers. We would like to secure this open communication channel and fortify the trust relationship between mobile OS and user.

1.3 Contributions

This dissertation proposes practical solutions for the aforementioned problems. For each problem, we elaborate on the attack vector, assess its current status, and develop deployable solutions that can effectively resolve the attack surface and raise the awareness of the developers/users.

1.3.1 Understanding and Defending the Binder Attack Surface

We identified and studied more than 100 vulnerabilities on the Binder attack surface. We analyzed these vulnerabilities to find that most of them are rooted at a common confusion of where the actual security boundary is among system developers. We thus highlight the deficiency of testing only on client-side public APIs and argue for the necessity of testing and protection on the `Binder` interface — the actual security boundary. Specifically, we design and implement *BinderCracker*, an automatic testing framework that supports context-aware fuzzing and actively manages the dependency between transactions. It does not require the source codes of the component under test, is compatible with services in different layers, and performs 7x better than simple black-box fuzzing. We also call attention to the attack attribution problem for IPC-based attacks. The lack of OS-level support makes it very difficult to identify the culprit apps even for developers with adb access. We address this issue by providing an informative runtime diagnostic tool that tracks the origin, scheme, content, and parsing details of each failed transaction. This brings transparency into the IPC process and provides an essential step for other in-depth analysis or forensics.

1.3.2 Reducing Unregulated Aggregation Across App Usage Behaviors

We present a fresh perspective of unregulated aggregation, focusing on monitoring, characterizing and reducing the underlying linkability across apps. The cornerstone of our study is the *Dynamic Linkability Graph* (DLG) which tracks app-level linkability during runtime. We observed how DLG evolves on real-world users and identified real-world evidence of apps abusing IPCs and OS-level identifying information to establish linkability. Based on these observations, we propose a linkability-aware extension to current mobile operating systems, called `LinkDroid`, which provides runtime monitoring and mediation of linkability across different apps. `LinkDroid` is a client-side solution and compatible with the existing smartphone ecosystem. It helps end-users "sense" this emerging threat and provides them intuitive opt-out options.

1.3.3 Enabling Unlinkability of Mobile Apps in User-level

Though the threat of unregulated aggregation is prevalent and severe, existing mobile OS vendors may be unlikely to adopt changes/extensions that can improve the privacy of users. This is due to a conflict of interest because the entire mobile ecosystem is fueled by the abundance of user information. To bridge this gap, we propose `Mask`, the first user-level solution that allows the user to negotiate to what extent his behavior can be linked and aggregated. Specifically, `Mask` introduces a set of *private execution modes* that allow the users to maintain multiple isolated profiles for each app. Each app profile can be temporary, being recycled after each session, or enduring, persists across multiple sessions. By enabling the private execution modes which isolate app usages at this very source, `Mask` provides a client-side solution without requiring any change to the existing ecosystem.

1.3.4 Continuous Authentication for Voice Assistants

We developed `VAuth`, the first system that provides continuous and usable authentication for voice assistants. We design `VAuth` to fit in various widely-adopted wearable devices, such as eyeglasses, earphones/buds and necklaces, where it collects the body-surface vibrations of the user and matches it with the speech signal received by the voice assistant's microphone. `VAuth` guarantees that the voice assistant executes *only* the commands that originate from the voice of the owner. We have evaluated `VAuth` with 18 users and 30 voice commands and find it to achieve an almost perfect matching accuracy with less than 0.1% false positive rate, regardless of `VAuth`'s position on the body and the user's language, accent or mobility. `VAuth` successfully thwarts different practical attacks, such as replayed attacks, mangled voice attacks, or impersonation attacks. It also has low energy and latency overheads and is compatible with most existing voice assistants.

CHAPTER II

Understanding and Defending the Android Binder Attack Surface

2.1 Introduction

Android is the most popular smartphone OS and dominates the global market with a share of more than 82% [97]. By the end of 2015, the total number of Android devices surpassed 1.4 billion, and more than 1.6 million mobile apps were available in Google Play for download [43, 83]. The developers of these apps are not always trustworthy; many of them might be inexperienced, careless or even malicious. Therefore, proper isolation between apps and the system is essential for robustness and security.

To meet this requirement, apps in Android execute in the application sandboxes. They depend on Inter-Process Communications (IPCs) extensively to interact with the system and other apps. `Binder`, as the cornerstone of this IPC mechanism, has long been believed as one of the most secure/robust components in Android. However, during the past year, there have been multiple CVE (Common Vulnerabilities and Exposures) reports discussing attacks exploiting the `Binder` interface [24–27, 87]. Interestingly, none of these attacks tries to undermine the security of `Binder` driver, but instead use `Binder` only as an attack gateway (entry point). A careful examination of the attack surface has led us to the discovery of the fundamental cause of this attack vector: an attacker can directly inject faulty transactions into system services by manipulating the `Binder` interface, and hence bypass all client-side sanity checks. Theoretically, this should not be an issue — a system service should not hinge on the validity of client-side transactions, and should always be robust on its own. However, we found that this principle has frequently been overlooked in the implementation of many Android system services, which led us to the following questions: *why do system developers keep on making this seemingly simple oversight, and what can we do to help mitigate this problem?*

To answer these questions, we conduct the first in-depth analysis of this attack surface. Specifically, we studied more than 98 generic system services (by Google) and 72 vendor-specific services (by Samsung) in 6 Android versions, and identified 137 vulnerabilities (unique crashes de-duplicated across versions) on this surface. We analyzed 115 of them in Android source codes and found that sanity checks are most extensive around client-side public APIs, and quickly become sporadic/careless after this defense line. Specifically, RPC parameters that are not exposed via public APIs are frequently left unchecked and the underlying (de-)serialization process of these parameters is often unprotected. This suggests that there is a mis-conception of where the security boundary is for Android system services — many seem to assume the security/trust boundary to be at the client-side public APIs, and whatever happens thereafter is free from obstruction since they already belong to the system territory. This mis-conception is understandable since Android provides convenient and automatic code generation tools (AIDL) that at one side relieve the developers from writing their own IPC stack, but at the other side hide all the details about RPC and `Binder`. Thus, we argue for the necessity of introducing automatic testing and protection at the `Binder` surface, i.e., the actual security boundary.

All the vulnerabilities reported in this chapter are identified by *BinderCracker*, a precautionary testing framework we developed for `Binder`-based RPCs. *Binder-*

Cracker is a context-aware fuzzing framework that understands the input and output structure of each RPC transaction as well as the inter-dependencies between them. This is essential because many transactions require inputs of remote object handles which are output of other transactions and cannot be recorded in the form of raw bytes. Before fuzzing a transaction, *BinderCracker* will automatically replay all the transactions it depends on and generate the correct context. *BinderCracker* does not require source codes of the services under test and works for services in both the Java and native layers. Thus, it is readily compatible with both Android system services and vendor-specific services. *BinderCracker* achieves effective vulnerability discovery — it identified 7x more vulnerabilities than simple black-box fuzzing within the same amount of time. Furthermore, since *BinderCracker* understands the schema of each low-level RPC transaction, we can easily configure it to test high-level abstraction or protocol built on top of the `Binder` primitives, such as Intent communications or app-specific protocols.

To help mitigate this emerging attack surface, we need to eliminate potential vulnerabilities as early as possible in the development cycle. Specifically, we suggest the use of various precautionary testing techniques (including *BinderCracker*) before each product release. This can stop a large number of vulnerabilities from reaching the end-users. In fact, many severe vulnerabilities [24–27, 87] could have been avoided had *BinderCracker* been deployed. Notably, 60% of the vulnerabilities identified by *BinderCracker* still remain unfixed by the time they are found. We summarize these vulnerabilities and have already reported them to AOSP. Many of the vulnerabilities we identified are found to be able to crash the entire Android Runtime, while others can cause specific system services or system apps to fail. Some vulnerabilities have further security implications, and may result in permission leakage, privileged code execution, targeted or permanent Denial-of-Service (DoS).

In case vulnerabilities leak through precautionary testing into the deployment phase, we need runtime defenses on this attack surface. Here, we addressed the urgent problem of attack attribution for IPC-based attacks, which have not received enough attention from the security community. Due to the lack of OS-level support, it is extremely difficult to identify the culprit app even for developers with adb access, let alone for average users. This suggests that an attacker app can sabotage the system or crash other apps without being accused of, or may even blame others. For example, the attacker app can crash Android Runtime whenever the user opens a competitor app, creating the illusion that the competitor app is buggy. Similar attacks are not rare between businesses with stiff competition [103]. To address this issue, we proposed an informative runtime diagnostic tool. It maintains the sender, schema, content and parsing information for each ongoing transaction, in case a failure/attack happens. Whenever a system service fails when processing an incoming transaction, a detailed report with the transaction information will be generated and a visual warning will be prompted to the user. The reporting process can also be triggered by access to privileged APIs or abnormal permission request, to catch attacks that do not warrant a program crash. This brings transparency into IPC communications and constitutes an essential first step for other in-depth analysis or forensics.

This work makes the following contributions: we

- Provide a systematic analysis of the attack surface by conducting security and root cause analysis on 100+ vulnerabilities. We summarized the common mistakes made by system developers and found the attack surface persists largely due to a common confusion of where the actual trust boundary is;

- Design and implement, *BinderCracker*, a context-aware fuzzing framework for Android IPCs that actively managed the dependencies between transactions. *BinderCracker* is compatible with Android system services, vendor-specific services and can be easily configured to fuzz high-level abstractions or protocols. It identifies 7x more vulnerabilities than a simple black-box fuzzing approach;

- Propose a system-level diagnostic tool which addresses the attack attribution problem for IPC-based attacks. By tracking the origin, schema and content of ongoing transactions, it brings transparency into Android IPCs and provides an essential step towards in-depth runtime analysis and defense.

The rest of the chapter is organized as follows. Section 2.2 summarizes related work in the field of software testing and Android security. Section 2.3 introduces `Binder` and AIDL in Android, and describes how Android uses these to build system services. Section 2.4 examines the attack surface and focus on explaining what mistakes have the developers made and why. Section 2.5 details the design and implementation of our testing framework, *BinderCracker*. Section 2.6 gives a comprehensive discussion on how to mitigate this attack surface with a special focus on the attack attribution problem. Section 2.7 discusses the insight, and other potential use-cases of *BinderCracker*, and finally, the chapter concludes with Section 2.8.

2.2 Related Work

Discussed below is related work in the field of software testing and Android security.

Software Testing In the software community, robustness testing falls into two categories: *functional* and *exceptional* testing. Functional testing focuses on verification of the functionality of software using expected input, while exceptional testing tries to apply unexpected and faulty inputs to crash the system. Numerous efforts have been made in the software testing community to test the robustness of Android [4, 5, 51, 65, 73, 112]. Most of them focus on the functional testing of GUI elements [4, 5, 51, 65]. Some conducted exceptional testing on the evolving public APIs [73]. In this work, we highlight the deficiency of testing public APIs only and conduct exceptional testing on lower-level `Binder`-based RPC interfaces.

Android Security Android has received significant attention from the research community as an open source operating system [17, 34, 35, 48, 69, 90, 94, 99]. Existing Android security studies largely focus on the imperfection of high-level permission model [37, 39, 80], and the resulting issues, such as information leakage [34], privilege escalation [17, 90] and collusion [69]. We will highlight the insufficient protection of Android's lower-level `Binder`-based RPC mechanism and how it affects the robustness of system services.

There also exist a few studies focusing on the IPC mechanism of Android [20, 32, 54, 66, 91]. However, they largely focus on one specific instance of Android IPC — Intent. Since the senders and recipients of Intents are both apps, manipulating Intents will not serve the purpose of exposing vulnerabilities in system services. Some researchers also made recommendations for hardening Android IPCs [54, 66] and pointed out that the key problem in Intent communication is the lack of formal schema. We demonstrate that even for mechanisms enforcing a formal schema, such as AIDL, robustness remains as a critical issue. There have also been some parallel attempts on fuzzing the `Binder` interface in industry [41, 47]. However, they focused on the technical details of implementing Proof-of-Concept (PoC) exploits on this interface and only tested simple fuzzing techniques. Our work focuses instead on understanding of the origin of the `Binder` attack surface and proposes practical defenses. We summarize the common mistakes made by system developers by studying 100+ real vulnerabilities and address the urgent problem of attack attribution for IPC-based attacks. Moreover, our context-aware fuzzing framework, *BinderCracker*, is sophisticated and works much more effectively than simple black-box fuzzing.

2.3 Android IPC and Binder

Android executes apps and system services as different processes and enforces isolation between them. To enable different processes to exchange information with each other, Android provides, `Binder`, a secure and extensible IPC mechanism. Described below are the basic concepts in the `Binder` framework and an explanation of how a typical system service is built using these basic primitives.

2.3.1 Binder

In Android, `Binder` provides a message-based communication channel between two processes. It consists of (i) a kernel-level driver that achieves communication across process boundaries, (ii) a `Binder` library that uses `ioctl` syscall to talk with the kernel-level driver, and (iii) upper-level abstracts that utilize the `Binder` library. Conceptually, `Binder` takes a classical client–server architecture. A client can send a transaction to the remote server via the `Binder` framework and then retrieves its response. The parameters of the transaction are marshalled into a `Parcel` object which is a serializable data container. The `Parcel` object is sent through the `Binder` driver and then gets delivered to the server. The server de-serializes the parameters of the `Parcel` object, processes the transaction, and returns a response in a similar way, back to the client. This allows a client to achieve *Remote Procedure Call* (RPC) and invoke methods on remote servers as if they were local. This `Binder`-based RPC is one of the most frequent forms of IPC in Android, and underpins the implementation of most system services.

2.3.2 Android Interface Description Language (AIDL)

Many RPC systems use IDL (*Interface Description Language*) to define and restrict the format of a remote invocation [66], and so does Android. The AIDL (*Android Interface Description Language*) file allows the developer to define the RPC interface both the client and the server agree upon [7]. Android can automatically generate Stub and Proxy classes from an AIDL file and relieve the developers from (re-)implementing the low-level details to cope with native `Binder` libraries. The

```
interface IQueueService {
  boolean add(String name);
  String peek();
  String poll();
  String remove();
}
```

Figure 2.1: An example AIDL file which defines the interface of a service that implements a queue.

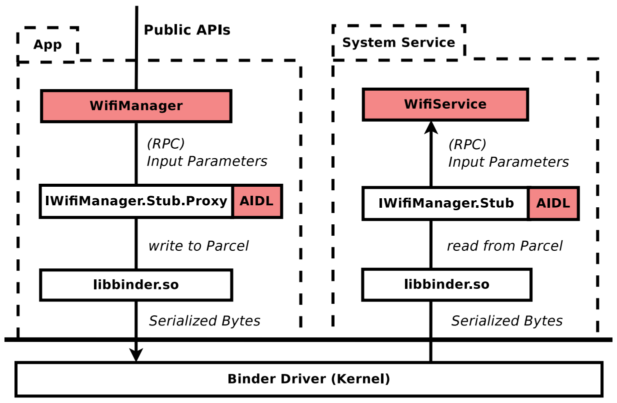


Figure 2.2: How does an app communicate with a system service using `Binder`-based RPC (using Wi-Fi service as an example)? The red shaded region represents the codes that need to be provided/implemented by the service developer.

auto-generated Stub and Proxy classes will ensure that the declared list of parameters will be properly serialized, sent, received, and de-serialized. The developer only needs to provide a `.aidl` file and implement the corresponding interface. In other words, the AIDL file serves as an explicit contract between client and server. This enforcement makes the `Binder` framework extensible, usable, and robust. Fig. 2.1 shows an example AIDL file that defines the interface of a service that implements a queue.

2.3.3 System Service

We describe next how the low-level concepts in the `Binder` framework are structured to deliver a system service, using Wi-Fi service as an example. To implement the Wi-Fi service, system developers only need to define its interfaces as an AIDL description, and then implement the corresponding server-side logic (`WifiService`) and client-side wrapper (`WifiManager`) (see Fig. 2.2). The serialization, transmission, and de-serialization of the interface parameters are handled by the codes automatically generated from the AIDL file. Specifically, when the client invokes some RPC method in the client-side wrapper `WifiManager`, the Proxy class `IWifiManager.Stub.Proxy` will marshall the input parameters in a `Parcel` object and send it across the process boundary via the `Binder` driver. The `Binder` library at the server-side will then unmarshall the parameters and invoke the `onTransact` function in the Stub class `IWifiManager.Stub`. This eventually invokes the service logic programmed in `WifiService`. Fig. 2.2 provides a clear illustration of the entire process.

2.4 Binder: The Attack Surface

The `Binder` driver serves as the boundary between two communicating parties and separates them as client and server. Existing attacks on this interface typically involve direct injection of a crafted transaction via the `Binder` interface. In theory, at which layer is a transaction injected at the client-side should not affect the security of the Android system — the server-side should always be robust on its own. This is probably a best engineering practice for any system that adopts a client–server model. However, as we will show later, this guideline is found to be frequently overlooked in the implementation of Android system services. Here, we review 100+ vulnerabilities found in six major Android versions and try to summarize the mistakes made by the system developers that turn the `Binder` interface into a tempting attack surface, and why system developers keep making these seemingly simple mistakes. All of the vulnerabilities discussed in this section are discovered by *BinderCracker*, the first context-aware fuzzing framework for Android IPCs. We will detail the design of *BinderCracker* in the next section.

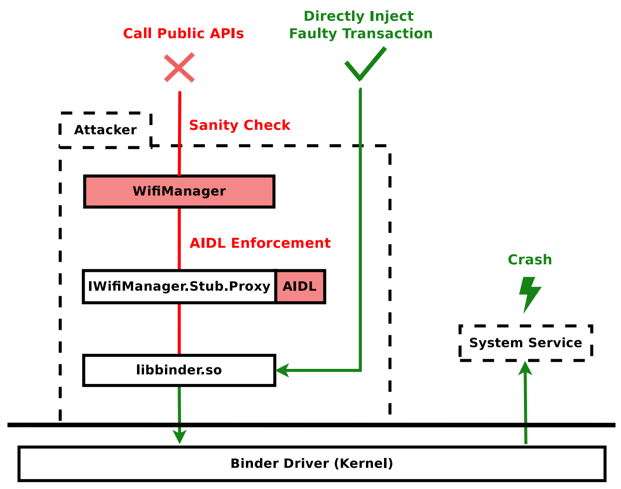


Figure 2.3: A typical attack scenario. By injecting faulty transactions via the `Binder` driver, an attacker can bypass the sanity check on public API and AIDL enforcement, and directly challenge the server-side.

2.4.1 Attack Model

We assume the adversary is a malicious app developer trying to sabotage the robustness or the integrity of Android system services. A system service can be generic, existing in Android framework base, or vendor-specific, introduced by device manufacturers. The attacker launch attacks by directly injecting crafted transactions into the `Binder` interface. The consequences can range from Denial-of-Service (DoS) attack to privileged code execution, depending on the payload and the target of the malicious transaction. We assume the attacker has no root permission and cannot break the security of OS kernel. Fig. 2.3 illustrates a typical attack scenario.

2.4.2 Overview of Vulnerabilities

We identified 137 vulnerabilities on the `Binder` attack surface by testing 6 major versions of Android: 4.1 (JellyBean), 4.2 (JellyBean), 4.4 (KitKat), 5.0 (Lollipop), 5.1 (Lollipop) and 6.0 (Marshmallow). Note that the number of discovered vulnerabilities have already been de-duplicated across versions. Specifically, we examined more than 98 generic system services (by Google) and 72 vendor-specific services (by

Version	*API*	*Market*	*Device*	*Build #*
4.1.1	16	9.0%	Galaxy Note 2	JRO03C
4.2.2	17	12.2%	Galaxy S4	JDQ39
4.4.2	19	36.1%	Galaxy S4	KOT49H
5.0.1	21	16.9%	Nexus 5	LRX22C
5.1.0	22	15.7%	Nexus 5	LMY47I
6.0.0	23	0.7%	Nexus 5	MRA58K

Figure 2.4: List of Android ROMs we tested using *BinderCracker*.

Samsung), which covers more than 2400 low-level RPC methods. The majority of the vulnerabilities are in Android framework while 15 of them are in vendor-specific (Samsung) services. All our experiments are conducted by running *BinderCracker* on official firmwares from major device manufacturers (see Fig. 2.4). An official firmware went through extensive testing by the vendors and is believed to be ready for a public release. Each firmware is tested in the initial state, right after it is installed. We didn't install any third-party app or change any configuration except for turning on the adb debugging option, ruling out the influence of external factors.

An RPC method is found to be *vulnerable* if testing it resulted in a fatal exception, crashing part of, or the entire Android Runtime. Each unique crash (stack traces) under an RPC interface is further referred to as an *individual vulnerability*. For each vulnerability reported here, we followed the process of: 1) identify it on an official ROM, 2) manually confirm that it can be reproduced, and 3) inspect the source codes for a root cause analysis. For vendor-specific vulnerabilities of which source codes are not available, such as many of the customized system services provided by Samsung, we only record the stack trace. Fig. 2.5 list the number of vulnerabilities grouped by the exception types in the crash traces. The security implications of the identified vulnerabilities will be further reviewed in Section 2.6.

Level	*Exception Type*	*Count*
Java	NullPointerException	29
	IllegalArgumentException	9
	OutOfMemoryError	8
	RuntimeException	8
	IllegalStateException	4
	StackOverflowError	4
	UnsatisfiedLinkError	2
	ArrayIndexOutOfBoundsException	2
	OutOfResourcesException	1
	SecurityException	1
	StringIndexOutOfBoundsException	1
	IOException	1
	BadParcelableException	1
Native	SEGV_MAPPER	31
	SI_TKILL	29
	BUS_ADRALN	4
	SEGV_ACCERR	1
	SI_USER	1

Figure 2.5: Vulnerabilities grouped by types of exception.

2.4.3 Root Cause Analysis

The direct causes of crashes are uncaught exceptions such as `NullPointerException` or `SEGV_MAPPER`, but the fundamental cause behind them is deeper. For each crashed system service of which source codes are available, we looked into the source codes and analyzed the root causes of the vulnerabilities. We noticed that sanity checks are most extensive around client-side public APIs, but are sporadic/careless after this line. This suggests that many system developers only considered the exploitation of public APIs, thus directly injecting faulty transactions to the `Binder` driver creates many scenarios that are believed to be 'unlikely' or 'impossible' in their mindset. Here, we highlight some of the new attack vectors enabled by attacking the `Binder` interface which contribute to most of the vulnerabilities we identified.

First, an attacker can manipulate RPC parameters that are not directly exposed via public APIs. For example, `IAudioFlinger` provides an RPC method

REGISTER_CLIENT. This method is only implicitly called in the Android middleware and is never exposed via public interfaces. Therefore, the developers of this system service may not expect an arbitrary input from this RPC method and didn't perform a proper check of the input parameters. In our test, sending a list of null parameters via the Binder driver can easily crash this service. This suggests that developers should not overlook RPC interfaces that are private or hidden.

Second, an attacker can bypass sanity checks around the public API, no matter how comprehensive they are. For example, the IBluetooth service provides a method called registerAppConfiguration. All of the parameters of this RPC method are directly exposed via a public API and there are multiple layers of sanity check around this interface. Therefore, if there is an erroneous input from the public API, the client will throw an exception and crash without even sending the transaction to the server side. However, using our approach, an attack transaction is directly injected to the Binder driver without even going through these client-side checks. This suggests that the server should always double-check input parameters on its own.

Third, an attacker can exploit the serialization process of certain data types and create inputs that are hazardous at the server side. For example, RemoteView is a Parcelable object that represents a group of hierarchical views. It contains a loophole in its de-serialization module which can cause a StackOverflow exception. As shown in Fig. 2.6, a bad recursion will occur if the input Parcel object follows a certain pattern. By directly manipulating the serialized bytes of the Parcel sent via the Binder driver, this loophole can be triggered and crash the server. This suggests that RPC methods with serializable inputs require special attention and sanity check is also essential in the de-serializaiton process.

These common mistakes made by system developers indicate there is a misconception of where the security boundary is for Android system services — many may assume the security/trust boundary is at the client-side public APIs, and whatever

```
android.widget.RemoteViews

private RemoteViews(Parcel parcel, BitmapCache bitmapCache) {

    int mode = parcel.readInt();

    ...

    if (mode == MODE_NORMAL) {
      ...
    } else {
      // recursively calls itself
      mL = new RemoteViews(parcel, mBitmapCache);
      // recursively calls itself
      mP = new RemoteViews(parcel, mBitmapCache);
      ...
    }

    ...
}
```

Figure 2.6: The constructor of the `RemoteView` class contains a loophole which can cause a `StackOverflow` exception. Specifically, a bad recursion will occur if the input Parcel object follows a specific pattern.

happens thereafter is free from obstruction since it is already in the system zone. This mis-conception is understandable since Android provides the convenient abstraction of AIDL and automatically generate codes that serialize, send, receive, de-serialize RPC parameters. This at one side relieves the developers from implementing their own IPC stack, but at the other side hide all the details about RPC and `Binder`. In other words, even though Android system services depend on IPC extensively, the developers are likely to be agnostic of that. Therefore, we advocate the importance of introducing automatic testing and protection at the `Binder` surface — the actual security boundary.

2.5 Effective Vulnerability Discovery

In this section, we explain how to conduct effective vulnerability discovery through the `Binder` interface. A naive approach is to issue transactions containing random bytes, also known as black-box fuzzing. In our experiment, we found that when using black-box fuzzing, almost all of the vulnerabilities are found within the first few fuzzing transactions, and a longer fuzzing time did not lead to the discovery of new bugs. The deficiency of black-box fuzzing is largely due to the extremely large and complex fuzzing space of this attack surface, and drives us to develope more sophisticated fuzzing techniques.

2.5.1 Unique Challenges

There are some unique challenges in fuzzing the `Binder` surface. First, a `Binder` transaction may contain non-primitive data types which result in complicated and hierarchical input schema. This affects 48% of all `Binder`-based RPCs and makes it difficult to fuzz according to parameter types. Moreover, the list of parameters included in a transaction is often dynamic instead of static. For example, when a transaction takes a `Bundle` object as input, what this object contains highly depends on the runtime status, and cannot be simply captured by a static interface description. All of these make it difficult to generate schemas of `Binder`-based transactions in an effective and reliable way.

Second, inter-dependencies often exist across `Binder` transactions. We found that 37% user-level RPC invocations require an input parameter that is the output of previous transactions. Note that, these input parameters are remote handlers that cannot be recorded in the form of raw bytes and can only be generated by executing the same transactions. This process is extremely crucial if we want to fuzz dynamically generated system services. While we can directly retrieve the handler of a statically cached system service and start to fuzz it, the handler of dynamically generated system services can only be retrieved by replaying the sequence of transactions it depends on.

Besides these two challenges, our design options are also restricted by the (un)availability of the source codes for the system services under test. For example, in a typical Samsung Galaxy device, there are more than 70 vendor-specific system services, the source codes of which are clearly unavailable. Even for system services that are included in the core Android framework, their source codes are typically proprietary when related to crypto or interactions with OEM hardwares. The scenario becomes more exaggerated if considering services exported by system or user-level apps.

2.5.2 BinderCraker: Design Overview

We present an overview of the design of *BinderCracker*. Our design addresses the above challenges by adopting a context-aware, replay-based approach that actively manages the dependencies across transactions. Specifically, *BinderCracker* includes a recording component, implemented as an Android extension (custom ROM), and a fuzzing component, implemented as a user-level app. The recording component collects detailed information of different `Binder` transactions and the fuzzing component tries to replay and mutate each recorded transaction for fuzzing purposes.

The recording component builds and records the schema (parameter types and structure) of each transaction during runtime by instrumenting the (de)-serialization process of the `Binder` transaction. This is different from the approach that parses RPC interface (AIDL) files and does not require access to the source codes. Specifically, it monitors and records how each parameter is unmarshalled from the `Parcel` object. This instrumentation works recursively with the (de)-serialization functions, and thus can understand and record complicated, hierarchical schemas and non-primitive types. In additional to recording the transaction schema, *BinderCracker* au-

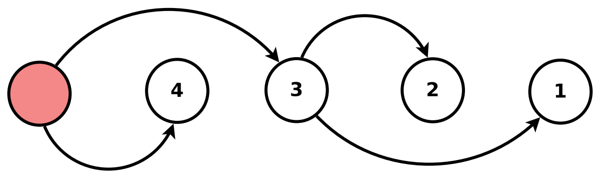


Figure 2.7: When fuzzing a transaction, we need to replay the supporting transactions according to their relative order in the dependency graph. This way, all the remote objects this transaction requires will be reconstructed during runtime.

tomatically matches the inputs and outputs of adjacent transactions and constructs a dependency graph among them. This dependency graph captures the sequence (tree) of transactions required to generate the inputs of a target transaction (see Fig. 2.7). This allows *BinderCracker* to automatically manage the dependencies between transactions and reconstruct the dynamic context each transaction requires. For example, certain system services, such as `IGraphicBufferProducer`, are dynamically initialized instead of statically cached in the `system_server`. Fuzzing it typically requires manually writing a code section that first generates this service and then using reflections on private APIs to retrieve the handler, which is tedious and not scalable. *BinderCracker* greatly simplifies this process since all the transactions that generate this service will be replayed automatically before the actual fuzzing process.

After recording the seed transactions and their dependencies, we need to utilize them to fuzz a system service. The fuzzing component of *BinderCracker* has a replay engine built-in and is implemented as a user-level app. Basically, it is manipulating (either directly or indirectly) a `binder_transaction_data` struct sent to the `Binder` driver. This data struct contains three important pieces of information we need to modify to send a fuzzing transaction and has the format as shown in Fig. 2.8. The `target.handle` field specifies the service this transaction is sent to. The `code` field represents a specific RPC method we want to fuzz. The `data` struct contains the serialized bytes of the list of parameters for the RPC method, which is inherently a `Parcel` object. `Parcel` is a container class that provides a convenient set of serialization and de-serialization methods for different data types. Both the client and the

```
struct binder_transaction_data {
  union {
    size_t handle; // (1).target service
    void *ptr;
  }target;
  void *cookie;
  unsigned int code; // (2).RPC method
  unsigned int flags;
  pid_t sender_pid;
  uid_t sender_euid;
  size_t data_size;
  size_t offsets_size;
  union {
        struct {

           binder_uintptr_t buffer;
           binder_uintptr_t offsets;
        } ptr;
        __u8 buf[8];
    } data; // (3).transactional data
};
```

Figure 2.8: The data struct sent through the `Binder` diver via the `ioctl` libc call. This struct contains three important pieces of information we need to modify to send a fuzzing transaction.

server work directly with this `Parcel` object to send and receive the input parameters. Later in this section, we will elaborate on how to modify the `handle` and `code` variables to redirect the transaction to a specific RPC method of a specified service, and how to fuzz the `Parcel` object to facilitate testing with different policies.

2.5.3 Transaction Redirection

There is a one-to-one mapping from the `handle` variable in the `binder_transaction_data` object to system service. This mapping is created during runtime and maintained by the `Binder` driver. Since the client has no control over the `Binder` driver, it cannot get this mapping directly. For system services that are statically cached, we can get them indirectly by querying a static service manager which has a fixed `handle` of 0. This service manager is a centralized controller for service registry and will be started before any other services. By sending a service interface descriptor (such as android.os.IWindowManager) to the service manager, it will return an `IBinder` object which contains the `handle` for the specified service. For system services that are dynamically allocated, we can retrieve them by recursively replaying the supporting transactions that generate these services (see Fig. 2.7).

After getting the `handle` of a system service, we need to specify the `code` variable in the `binder_transaction_data` object. Each code represents a different RPC method defined in the AIDL file. This mapping can be found in the Stub files which are automatically generated from the AIDL file. The `code` variable typically ranges from 1 to the total number of methods declared in the AIDL file. For native system services that are not implemented in Java, this mapping is directly coded in either the source files or the header files. Therefore, we scan both the AIDL files and the native source codes of Android to construct the mapping between transaction codes and RPC methods.

2.5.4 Transaction Fuzzing

After being able to redirect a `Binder` transaction to a chosen RPC method of a chosen system service, the next step is to manipulate the transaction data and create faulty transactions that are unlikely to occur in normal circumstances. Here, *BinderCracker* utilize our context-aware replay engine to generate semi-valid fuzzing transactions. A transaction is said to be *semi-valid* if all of the parameters it contains are valid except for one. Semi-valid transactions can dive deeper into the program structure without being early rejected, thus is able to reveal more in-depth vulnerabilities.

In summary, *BinderCracker* maintains both the type hierarchy and dependency graph when recording a seed transaction. These information capture the semantic and context of each transaction and help *BinderCracker* generate semi-valid fuzzing

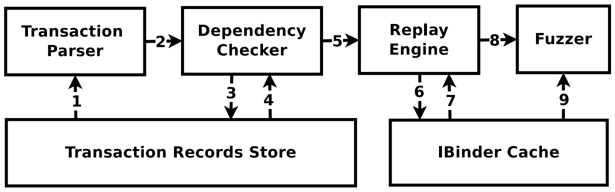


Figure 2.9: How does *BinderCracker* generate semi-valid fuzzing transactions from seed transactions.

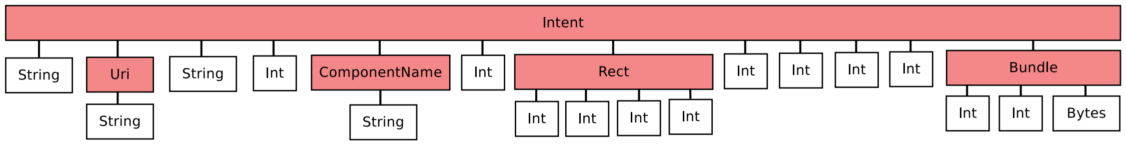


Figure 2.10: The internal type structure of a non-primitive data type, `Intent`, generated by recording the de-serialization process of each non-primitive type. Note that this type structure is dynamic — it depends on what has been put into this `Intent` during runtime.

transactions. Specifically, it follows the process illustrated in Fig. 2.9. For each seed transaction we want to fuzz, we first parse the raw bytes of the transaction and unmarshall non-primitive data types into an array of primitive types (step 1). This utilizes the type hierarchy recorded with the seed transaction. Then, we check the dependency of the transaction (step 2) and retrieve all the supporting transactions (steps 3, 4). This step utilizes the dependency graph recorded with the seed transaction. After that, we need to replay the supporting transactions (step 5) to generate and cache the remote `IBinder` object handles (steps 6, 7). Finally, the fuzzer can start to generate semi-valid fuzzing transactions by mutating each parameter in the seed transaction according to their data types (steps 8, 9). For example, for numerical types such as Integer, we may add or substrate a small delta from the current value or change it to Integer.MAX, 0 or Integer.MIN; for literal types such as String, we may randomly mutate the bytes contained in the String, append new content at start or end, or insert special characters at certain locations.

After sending a faulty transaction to a remote service, there are a few possible responses from the server-side. First, the server detects the input is invalid and rejects the transaction, writing an `IllegalArgumentException` message back to the client.

Second, the server accepts the argument and starts the transaction, but encounters unexpected states or behaviors and catches some type of `RuntimeException`. Third, the server doesn't catch some bizarre scenarios, causes a Fatal Exception and crashes itself. In this work, we focus on the last type of responses, as it is most critical and has disastrous consequences.

2.5.5 Experimental Results

We collected more than one million valid seed transactions by running 30 popular apps in two latest Android versions (Android 5.1 and Android 6.0). For each RPC interface, we sampled the transactions and selected those with unique transaction schema/structures. Based on this seed dataset, we performed fuzzing test on more than 445 RPC methods of 78 system services. In total, we identified 89 vulnerabilities in Android 5.1 and Android 6.0 which is 7x more than simple fuzzing with the same time spent. Compared to the vulnerabilities identified using simple black-box fuzzing, the vulnerabilities exposed by context-aware fuzzing are more interesting and have severer security implications — we start to identify buffer overflow, serialization bugs in deeper layers of the code, instead of just simple crashes or parsing errors (see Section 2.6 for details).

Moreover, since the approach *BinderCracker* adopts is generic, we can easily configure it to fuzz higher-level abstractions or protocols. For example, `Intent` is a high-level abstraction built on top of the `Binder` RPCs. It is used as a user-level communication primitive to launch apps, services or trigger broadcast receivers. With some simple configuration, we can turn *BinderCracker* into an `Intent` fuzzer. Specifically, we configure *BinderCracker* to only fuzz three RPC interfaces that the `Intent` communication mechanism is built upon, and only mutate the `Intent` parameter in these RPC calls. This makes *BinderCracker* a useful `Intent` fuzzer that automatically tracks and utilizes the internal type hierarchy of `Intent extras (Bundle)`. Fig. 2.10

illustrates the input structure of an example Intent we fuzz. In total, *BinderCracker* identified more than 20 vulnerabilities in the Intent communication, many of which exist in the de-serialization process of Intent.

2.6 Defenses

New vulnerabilities are still emerging on the `Binder` attack surface whenever there is a major upgrade of the Android code base. This is because, considering the code size of Android, it is almost impossible to prevent the developers from writing buggy codes. We discussed how to conduct effective precautionary testing to help expose vulnerabilities before releasing the new ROM, and also explained why it is very difficult, if not impossible, to conduct runtime defense in existing Android systems. The major obstacle in developing runtime defenses is the lack of transparency/auditing on IPC transactions. When a system service fails, no one knows why it crashed and who caused its crash without proper OS-level support. Thus, a system-level IPC diagnosis tool is essential for any in-depth runtime analysis, such as attack attribution and cross-device analytics.

2.6.1 Precautionary Testing

Before releasing a new ROM, developers can conduct precautionary testing. The defense can be done early, in the development phase of each system service, or later, after the entire ROM gets built.

Android has already adopted a static code analysis tool, `lint`, to check potential bugs and optimizations for correctness, security, performance, usability, accessibility and internationalization [52]. Specifically, `lint` provides a feature that supports inspection with annotations. This allows the developer to add metadata tags to variables, parameters and return values. For example, the developer can mark an input parameter as `@NonNull`, indicating that it cannot be `Null`, or mark it as

`@IntRange(from=0,to=255)`, enforcing that it can only be within a given range. Then, `lint` automatically analyzes the source codes and prompts potential violations. This can be extended to support inspections of RPC interfaces, allowing developers to explicitly declare the constraints for each RPC input parameter. This way, many potential bugs can be eliminated during the development phase. This defense is practical and comprehensive but requires system developers to specify the metadata tags for each RPC interface.

We can also conduct precautionary testing during runtime after the ROM has been built. Our system, *BinderCracker*, is effective in identifying vulnerabilities and can be used as an automatic testing tool. By fuzzing various system services with different policies, a large number of vulnerabilities can be eliminated before reaching the end-users. Actually, many severe vulnerabilities [24–27, 87] could have been avoided if a tool like *BinderCracker* had been deployed. Note that the effectiveness of *BinderCracker* depends on the quality and coverage of the seed transactions. Besides collecting execution traces of a large number of apps, another potential way of generating a comprehensive seed dataset is to incorporate the functional unit tests of each system service.

2.6.2 Security Implications

Most of the vulnerabilities *BinderCracker* discovered (more than 90%) can be used to launch a Denial-Of-Service (DoS) attack. Some of them are found to be able to crash the entire Android Runtime, while others can cause specific system services or system apps to fail. Fig. 2.11 shows the distribution of the affected services (apps). When launching a DoS attack, the attacker can trigger a crash either consistently or only under certain conditions, for example, when a competitor's app is running. This can create the impression that the competitor's app is buggy and unusable. We even identified multiple vulnerabilities (in the de-serialization process of `Intent`)

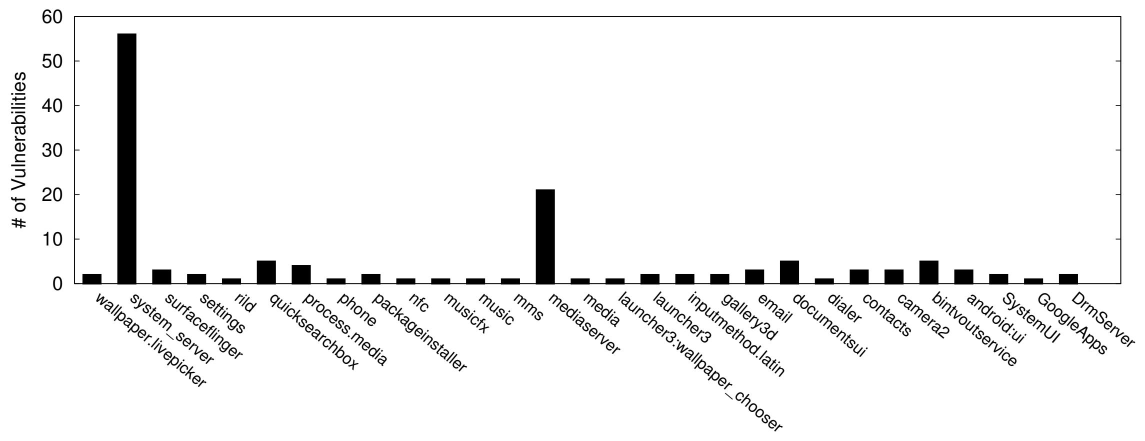


Figure 2.11: Many of the vulnerabilities we identified are found to be able to crash the entire Android Runtime (system_server), while others can cause specific system services (mediaserver) or system apps (nfc, contacts, etc) to fail.

that can cause targeted crash of almost any system/user-level apps, without crashing the entire system. Specifically, an attacker can craft an Intent that contains a malformated `Bundle` object and send to the target app. This will cause a crash during the de-serialization process of the `Intent` object before the target app can conduct any sanity check. Moreover, it can be very challenging to identify the attacker app under these scenarios because the OS only knows which service/app is broken, but cannot tell who crashed it. We will discuss more about the attack attribution process in later sections.

We also discovered a few vulnerabilities that can cause other serious security problems. We found that in several RPC methods, the server-side fails to check potential Integer overflows. This may lead to disastrous consequences when exploited by an experienced attacker. For example, in `IGraphicBufferProducer` an Integer overflow exists such that when a new `NativeHandle` is created, the server will malloc smaller memory than it actually requested (see Fig. 2.12). Subsequent writes to this data struct will corrupt the heap on the server-side. This vulnerability has been demonstrated to be able to achieve privileged code execution, and insert any arbitrary code into `system_server` [25]. We also found a vulnerability in `IContentService`

```
native_handle_t* native_handle_create(int numFds, int numInts)
{
    // numFds & numInts are not checked!
    native_handle_t* h = malloc( ...
        + sizeof(int)*(numFds+numInts));

    h->version = sizeof(native_handle_t);
    h->numFds = numFds;
    h->numInts = numInts;

    return h;
}
```

Figure 2.12: The constructor of the `native_handle` has an Integer Overflow vulnerability that can cause a heap corruption on the server-side. This can lead to privileged code execution in `system_server`.

that can lead to an infinite bootloop, which can only be resolved by factory recovery or flashing a new ROM. This is also classified as High Risk according to the official specification of Android severity levels [92].

Besides RPC methods that are not well implemented, we also discovered RPC methods that are not properly protected by existing Permission models. In official ROMs of Samsung Galaxy 4 (Android 4.2.2 and Android 4.4.2), an attacker can reboot the device by directly sending a transaction to `PackageManagerService` via the `Binder` driver without requiring the REBOOT permission. This is critical since REBOOT is a sensitive permission only granted to system apps. The other service is `ICoverManager`, a customized service from Samsung. An attacker can invoke a certain RPC method of `ICoverManager` and block the entire screen with a pop-up blank Activity. The blank Activity cannot be revoked using any virtual or physical button and the only exit is restarting the device.

2.6.3 Vulnerabilities: Fixed and Unfixed

We examined how many of the vulnerabilities discovered by *BinderCracker* remain unfixed and are potentially zero-day when they are found. Our analysis is based on the public changes of the source codes across different Android versions and revisions. We skipped the 15 vulnerabilities in vendor-specific system services and 7 in generic system services due to the unavailability of source codes. Note that not all generic system services are open source, especially when it is related to decryption/encryption or interactions with OEM hardware.

Of the 115 analyzed vulnerabilities in Android code bases, only 18 have been fixed by adding additional sanity checks of input parameters. Another 12 vulnerabilities 'disappeared' during several major Android version upgrades either because 1) the corresponding source codes (or API) have been deleted; or 2) new updates in other parts of the source codes accidentally bypass the vulnerable source codes. For example, some crashes are caused by a recursive call in the `RemoteView` class (see Fig. 2.6). Similar crashes disappeared after Android 5.0. We looked into the source codes and found this is not because the bug has been fixed, but because in new versions of Android a faulty transaction will create an additional Exception before it reaches the vulnerable codes. The additional Exception is properly caught and accidentally avoids the fatal crash caused by the real vulnerability. We do not consider this as a 'fix' since an attacker can still recreate the crash by manually crafting a transaction which bypasses the new code updates. Fig. 2.13 illustrates the proportion of vulnerabilities that are fixed, disappeared and unfixed. We have already submitted all unfixed vulnerabilities to AOSP.

2.6.4 Runtime Diagnostics and Defenses

It will be helpful if Android can provide some real-time defense against potential vulnerabilities even after the ROM has been deployed on end-users' devices. Here, we

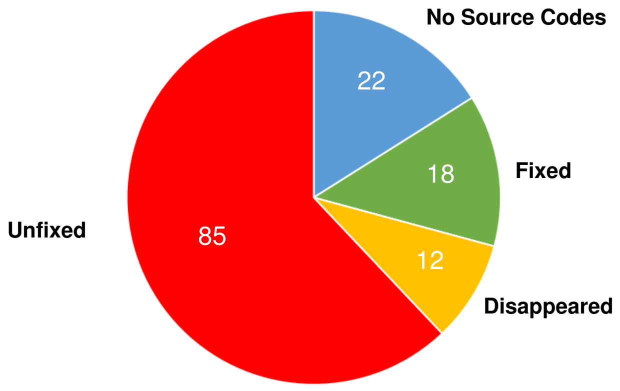


Figure 2.13: Number of the vulnerabilities that are fixed, disappeared and unfixed.

focus on specific defenses on the `Binder` layer, excluding generic defenses such as Address Space Layout Randomization (ASLR), SELinux, etc. They have been discussed extensively eslewhere [59, 93, 96] and are not specific to our scenario. Basically, there are two potential defenses one can provide on the `Binder` surface during runtime: (i) *intrusion detection/prevention*, identifying and rejecting transactions that are malicious, and (ii) *intrusion diagnostics*, making an attack visible after the transaction has already caused some damage. Unfortunately, both approaches are not applicable in existing Android systems. Next, we explain why the first approach is inherently challenging and then describe our efforts to enable the latter.

To provide runtime intrusion prevention, one needs to perform some type of abnormality detection on incoming transactions. This works by examining the input parameters of valid/invalid RPC invocations and characterizing the rules or boundaries. However, in our case, it is not practical for the following reasons. First, `Binder` transactions occur at a very high frequency but a mobile device has only limited energy and computation power. Second, parameters in `Binder` transactions are very diverse, codependent, and evolving dynamically during runtime, and hence clear boundaries or rules may not exist. Third, end-users are not likely to accept even the smallest false-positive rate. One can, of course, build a very conservative blacklist-based system and hard-coding rules of each potential vulnerability in the database. However,

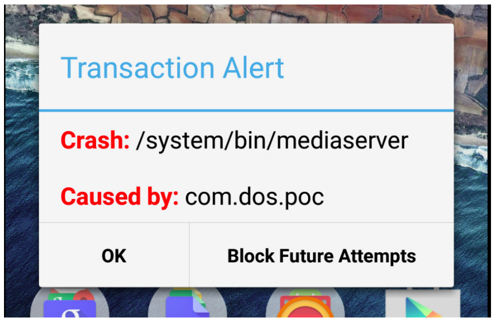
```
[Hash]: 1450375626446161
[Client-Head]-------------------------------------------
[0]:    android.media.IAudioFlinger     22
[1]:    pkg(com.example.sample)
[2]:    uid(10078)      pid(21225)
[3]:    Int32(0)String16(Int32(4)Raw(56@8))Int32(64)Int32(68)Int32(72)
[4]:    pos(64)
[5]:    76      Parcel(
0x00: 04004800 1b000000 61006e00 64007200  '..H.....a.n.d.r.'
0x10: 6f006900 64002e00 6d006500 64006900  'o.i.d...m.e.d.i.'
0x20: 61002e00 49004100 75006400 69006f00  'a...I.A.u.d.i.o.'
0x30: 46006c00 69006e00 67006500 72000000  'F.l.i.n.g.e.r...'
0x40: 00000000 01000000 10000000           '...........    ')
```

Figure 2.14: A detailed crash report which includes the system service under attack (0), transaction sender (1 and 2), schema (3), position of the parsing cursor (4) and raw content (5).

this seems unnecessary, especially when Android nowadays supports directly pushing security updates (patches) to devices of end-users.

An alternative solution is to diagnose, instead of prevent problems. It would be helpful if we can provide more visibility on how malicious transactions actually undermine a device. Even though this cannot stop the single device from being attacked, we can still utilize the collected statistics to develop in-time security patches, benefiting the vast majority of end-users. However, it is impossible to conduct informative diagnostic on the `Binder` layer even for developers with adb access. This is due to a lack of transparency/auditing on IPC transactions. Essentially, when a system service fails, no one knows why it crashed and who crashed it without proper OS-level support. This, also known as *attack attribution*, has not yet received enough attention from the research community. To fill this gap, we propose a system-layer diagnostic tool and demonstrate its use for more in-depth analyses.

The diagnostic tool is designed to provide three important functionalities: 1) when a service fails when processing an incoming transaction, the sender of the transaction will be recorded and the user will be warned with a visual prompt; 2) detailed information of the failed transaction, including the content, schema and parsing status will be dumped into a report for future forensics; 3) a signature of the transaction


Transaction Alert

Crash: /system/bin/mediaserver

Caused by: com.dos.poc

Figure 2.15: User receives a visual prompt in case of transaction failure and can choose to block it to counter continuous DoS attack.

will be generated and the user can review it and choose to block future occurrences of the same transaction. These, together, capture a snapshot of the IPC stack in case of a potential attack. This snapshot can be triggered by a crash, for DoS attacks, or by access to sensitive/privileged APIs, for privilege escalation attacks.

The diagnostic tool can be implemented in a similar way as the recording component of *BinderCracker*, by instrumenting the `Binder` framework and the `Parcel` class. We can further retrieve the sender of each transaction by calling `Binder.getCallingUid()` in the victim system service, and get the package name of the sender by querying the `PackageManagerService` with the retrieved uid. The same flow is also used to support permission checks in Android system services. The schema of each transaction is constructed and maintained during runtime by tracking the de-serialization process of the `Parcel`. Whenever an Exception is thrown and not caught by any of the Exception handling blocks, the transaction information maintained will be dumped as a separate report (see Fig. 2.14). The parsing information will be appended, indicating which parameters the service is parsing (have just parsed) when the failure/attack happens. The signature of the transaction will be recorded and the user will be prompted with a Notification (see Fig. 2.15). End-users can choose to block transactions with the same signature in the future to mitigate continuous DoS attacks. This marks an essential step towards more sophisticated runtime analysis, such cross-device analytics.

2.7 Discussion

We have assessed a risky attack surface comprehensively which has long been overlooked by the system developers of Android. As our experimental results demonstrated, new vulnerabilities are still emerging on this attack surface and *BinderCracker* can help eliminate potential vulnerabilities in future releases of Android. The lessons learned can transcend to other platforms facing similar issues, such as vehicular systems (CAN buses and ECUs), wearable devices, etc. We highlight that, although many systems adopt a client–server model in the design of their internal system components, they rarely follow the security standards of a real client–server model as in a networked environment. In many scenarios, a component may fall into the wrong hands and create serious security threats.

Our context-aware fuzzing is generic and not limited to system services. In fact, it can also be applied to services exported by user-level apps. For example, Facebook alone exports more than 30 services to other apps which forms a large attack surface. By performing fuzzing on this interface, more app-level vulnerabilities are expected to be unearthed. However, due to the unavailability of source codes, it is difficult to analyze the root causes and security implications of the identified vulnerabilities. Note that, although lack of source codes won't affect the discovery of vulnerabilities, it does make it more difficult to understand their implications.

2.8 Conclusion

In this chapter, we conducted an in-depth analysis on an emerging attack surface in Android. We summarized the common mistakes made by system developers that produces this attack surface and highlight the importance of testing and protection on the `Binder` interface, the actual trust boundary. We designed and implemented *BinderCracker*, a precautionary testing framework that achieves automatic vulnerability discovery. It supports context-aware fuzzing and performs 7x more effectively than a simple black-box fuzzing approach. We also addressed the urgent problem of attack attribution for IPC-based attacks, proposing OS-level runtime diagnostics support. These mechanisms are useful, practical and can be easily integrated into the development/deployment cycle of Android.

CHAPTER III

Reducing Unregulated Aggregation of App Usage Behaviors

3.1 Introduction

Mobile users run apps for various purposes, and exhibit very different or even unrelated behaviors in running different apps. For example, a user may expose his chatting history to WhatsApp, mobility traces to Maps, and political interests to CNN. Information about a single user, therefore, is scattered across different apps and each app acquires only a partial view of the user. Ideally, these views should remain as 'isolated islands of information' confined within each of the different apps. In practice, however, once the users' behavioral information is at the hands of the apps, it may be shared or leaked in an arbitrary way without the users' control or consent. This makes it possible for a curious adversary to aggregate usage behaviors of the same user across multiple apps without his knowledge and consent, which we refer to as *unregulated aggregation* of app-usage behaviors.

In the current mobile ecosystem, many parties are interested in conducting unregulated aggregation, including:

- *Advertising Agencies* embed ad libraries in different apps, establishing an explicit channel of cross-app usage aggregation. For example, Grindr is a geosocial app geared towards gay users, and BabyBump is a social network for expecting parents. Both apps include the same advertising library, MoPub, which can aggregate their information and recommend related ads, such as on gay parenting books. However, users may not want this type of unsolicited aggregation, especially across sensitive aspects of their lives.

- *Surveillance Agencies* monitor all aspects of the population for various precautionary purposes, some of which may cross the 'red line' of individuals' privacy. It has been widely publicized that NSA and GCHQ are conducting public surveillance by aggregating information leaked via mobile apps, including popular ones such as Angry Birds [8]. A recent study [108] shows that a similar adversary is able to attribute up to 50% of the mobile traffic to the "monitored" users, and extract detailed personal interests, such as political views and sexual orientations.

- *IT Companies* in the mobile industry frequently acquire other app companies, harvesting vast user base and data. Yahoo alone acquired more than 10 mobile app companies in 2013, with Facebook and Google following closely behind [2]. These acquisitions allow an IT company to link and aggregate behaviors of the same user from multiple apps without the user's consent. Moreover, if the acquiring company (such as Facebook) already knows the users' real identities, usage behaviors of all the apps it acquires become identifiable.

These scenarios of unregulated aggregation are realistic, financially motivated, and are only becoming more prevalent in the foreseeable future. In spite of this grave privacy threat, the process of unregulated aggregation is unobservable and works as a black box — no one knows what information has actually been aggregated and what really happens in the cloud. Users, therefore, are largely unaware of this threat and have no opt-out options. Existing proposals disallow apps from collecting user behaviors and shift part of the app logic (e.g., personalization) to the mobile OS or trusted cloud providers [29, 60]. This, albeit effective, is against the incentive of app developers and requires construction of a new ecosystem. Therefore, there is an urgent need for a practical solution that is compatible with the existing mobile ecosystem.

In this chapter, we propose a new way of addressing the unregulated aggregation problem by monitoring, characterizing and reducing the underlying linkability across apps. Two apps are *linkable* if they can associate their usage behaviors of the same user. This linkability is the prerequisite of conducting unregulated aggregation and represents an upper-bound of the potential threat. Researchers studied linkability under domain-specific scenarios, such as on movie reviews [79] and social networks [58]. In contrast, we focus on the linkability that is ubiquitous in the mobile ecosystem and introduced by domain-independent factors, such as device IDs, account numbers, location and inter-app communications. Specifically, we model mobile apps on the same device as a *Dynamic Linkability Graph* (DLG) which monitors apps' access to OS-level identifying information and cross-app communication channels. DLG quantifies the potential threat of unregulated aggregation and allows us to monitor the linkability across apps during runtime.

We implemented DLG as an Android extension and observed how it evolved on 13 users during a period of 47 days. The results reveal an alarming view of the app-level linkability in the wild. Two random apps (installed by the same user) are linkable with a probability of 0.81. Specifically, 86% of the apps a user installed are directly linkable to the Facebook app, namely, his real identity. In particular, we found that apps frequently abuse OS-level information and inter-process communication (IPC) channels in unexpected ways, establishing the linkability that is unrelated to app functionalities. For example, we found that many of the apps requesting account information collect all of the user's accounts even when they only need one to function correctly. We also noticed that some advertising agencies, such as Admob and Facebook, use IPCs to share user identifiers with other apps, completely bypassing system permissions and controls. Furthermore, we identified cases when different apps write and read the same persistent file in shared storage to exchange user identifiers. The end-users should be promptly warned about these unexpected behaviors to reduce unnecessary linkability.

Based on the above observations, we propose `LinkDroid`, a linkability-aware extension to Android which provides runtime monitoring and mediation of the linkability across apps. `LinkDroid` introduces a new dimension to privacy protection on smartphones. Instead of checking whether some app behavior poses direct privacy threat, `LinkDroid` warns users about how it implicitly affects the linkability across apps. Practicality is a main driver for the design of `LinkDroid`. It extends the widely-deployed (both runtime and install-time) permission model on the mobile OS that end-users are already familiar with. Specifically, `LinkDroid` provides the following privacy-enhancing features:

- **Install-Time Obfuscation:** `LinkDroid` obfuscates device-specific identifiers that have no influence on most app functionalities, such as IMEI, Android ID, etc. We perform this during install-time to maintain the consistency of these identifiers within each app.
- **Runtime Linkability Monitoring:** When an app tries to perform a certain action that introduces additional linkability, users will receive a just-in-time prompt and an intuitive risk indicator. Users can then exercise runtime access control and choose any of the opt-out options provided by `LinkDroid`.
- **Unlinkable Mode:** The user can start an app in unlinkable mode. This will create a new instance of the app which is unlinkable with other apps. All actions that may establish a direct association with other apps will be denied by default.

This way, users can enjoy finer-grained privacy protection, unlinking only a set of app sessions.

We evaluated `LinkDroid` on the same set of 13 users as in our measurement and found that `LinkDroid` reduces the cross-app linkability substantially with little loss of app performance. The probability of two random apps being linkable is reduced from 0.81 to 0.21, and the percentage of apps that are directly linkable to Facebook drops from 86% to 18%. On average, a user only needs to handle 1.06 prompts per day in the 47-day experiments and the performance overhead is marginal.

This work makes the following contributions:

1. Introduction of a novel perspective of defending against unregulated aggregation by addressing the underlying linkability across apps (Section 3.2).

2. Proposal of the Dynamic Linkability Graph (DLG) which enables runtime monitoring of cross-app linkability (Section 3.3).

3. Identification of real-world evidence of how apps abuse IPCs and OS-level information to establish linkability across apps (Section 3.4).

4. Addition of a new dimension to access control based on the runtime linkability, and development of a practical countermeasure, `LinkDroid`, to defend against unregulated aggregation (Section 3.5).

3.2 Privacy Threats: A New Perspective

In this section, we will first introduce our threat model of unregulated aggregation and then propose a novel perspective of addressing it by monitoring, characterizing and reducing the linkability across apps. We will also summarize the explicit/implicit sources of linkability in the current mobile app ecosystem.

3.2.1 Threat Model

In this work, we target unregulated aggregation across app-usage behaviors, i.e., when an adversary aggregates usage behaviors across multiple functionally-independent apps without users' knowledge or consent. In our threat model, an adversary can be any party that collects information from multiple apps or controls multiple apps, such as a widely-adopted advertising agency, an IT company in charge of multiple authentic apps, or a set of malicious colluding apps. We assume the mobile operating system and network operators are trustworthy and will not collude with the adversary.

3.2.2 Linkability: A New Perspective

There are many parties interested in conducting unregulated aggregation across apps. In practice, however, this process is unobservable and works as a black box — no one knows what information an adversary has collected and whether it has been aggregated in the cloud. Existing studies propose to disable mobile apps from collecting usage behaviors and shift part of the app logic to trusted cloud providers or mobile OS [29, 60]. These solutions, albeit effective, require building a new ecosystem and greatly restrict functionalities of the apps. Here, we address unregulated aggregation from a very different angle by monitoring, characterizing and reducing the underlying linkability across mobile apps. Two apps are *linkable* if they can associate usage behaviors of the same user. This linkability is the prerequisite of conducting unregulated aggregation, and represents an "upper-bound" of the potential threat. In the current mobile app ecosystem, there are various sources of linkability that an adversary can exploit. Researchers have studied linkability under several domain-specific scenarios, such as movie reviews [79] and social networks [58]. Here, we focus on the linkability that is ubiquitous and domain-independent. Specifically, we group its contributing sources into the following two fundamental categories.

Type	*2013-3*	*2013-10*	*2014-8*	*2015-1*
Android ID	80%	84%	87%	91%
IMEI	61%	64%	65%	68%
MAC	28%	42%	51%	55%
Account	24%	29%	32%	35%
Contacts	21%	26%	33%	37%

Table 3.1: Apps are increasingly interested in requesting persistent and consistent identifying information during the past few years.

OS-Level Information The mobile OS provides apps ubiquitous access to various system information, many of which can be used as consistent user identifiers across apps. These identifiers can be *device-specific*, such as MAC address and IMEI, *user-specific*, such as phone number or account number, or *context-based*, such as location or IP clusters. We conducted a longitudinal measurement study from March 2013 to January 2015, on the top 100 free Android apps in each category. We excluded the apps that are rarely downloaded, and considered only those with more than 1 million downloads. We found that apps are getting increasingly interested in requesting persistent and consistent identifying information, as shown in Table 3.1. By January 2015, 96% of top free apps request both the Internet access and at least one persistent identifying information. These identifying vectors, either explicit or implicit, allow two apps to link their knowledge of the same user at a remote side without even trying to bypass on-device isolation of the mobile OS.

Inter-Process Communications The mobile OS provides explicit Inter-Process Communication (IPC) channels, allowing apps to communicate with each other and perform certain tasks, such as export a location from Browser and open it with Maps. Since there is no existing control on IPC, colluding apps can exchange identifying information of the user and establish linkability covertly, without the user's knowledge. They can even synchronize and agree on a randomly-generated sequence as a custom user identifier, without accessing any system resource or permission. This problem gets more complex since apps can also conduct IPC implicitly by reading and writing shared persistent storage (SD card and databases). As we will show in Section 3.4, these exploitations are not hypothetical and have already been utilized by real-world apps.

3.3 Dynamic Linkability Graph

The cornerstone of our work is the Dynamic Linkability Graph (DLG). It enables us to monitor app-level linkability during runtime and quantify the linkability introduced by different contributing sources. In what follows, we will elaborate on the definition of DLG, the linkability sources it considers, and describe how it can be implemented as an extension of Android.

3.3.1 Basic Concepts

We model linkability across different apps on the same device as an undirected graph, which is called the *Dynamic Linkability Graph* (DLG). Nodes in DLG represent apps and edges represent linkability introduced by different contributing sources. DLG monitors the linkability during runtime by tracking the apps' access to various OS-level information and IPC channels. An edge exists between two apps if they accessed the same identifying information or engaged in an IPC. Fig. 3.1 presents an illustrative example of DLG.

DLG presents a comprehensive view of the linkability across all installed apps. An individual adversary, however, may only observe a subgraph of the DLG. For example, an advertising agency only controls those apps (nodes) that incorporate the same advertising library; an IT corporate only controls those apps (nodes) it has already acquired. In the rest of the chapter, we focus on the generalized case (the entire DLG) instead of considering each adversary individually (subgraphs of DLG).

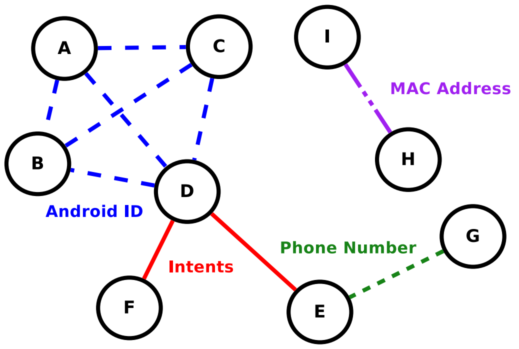


Figure 3.1: An illustrative example of DLG. Edges of different types represent linkability introduced by different sources.

3.3.2 Definitions and Metrics

Linkable Two apps a and b are *linkable* if there is a path between them. In Fig. 3.1, app A and F are linkable, app A and H are not linkable.

Gap is defined as the number of nodes (excluding the end nodes) on the shortest path between two linkable apps a and b. It represents how many additional apps an adversary needs to control in order to link information across a and b. For example, in Fig. 3.1, $gap_{A,D} = 0$, $gap_{A,E} = 1$, $gap_{A,G} = 2$.

Linking Ratio (LR) of an app is defined as the number of apps it is linkable to, divided by the number of all installed apps. LR ranges from 0 to 1 and characterizes to what extent an app is linkable to others. In DLG, LR equals to the size of the *Largest Connected Component* (LCC) this app resides in, excluding itself, divided by the size of the entire graph, also excluding itself:

$$LR_a = \frac{size(LCC_a) - 1}{size(DLG) - 1}$$

Linking Effort (LE) of an app is defined as the *Linking Effort* (LE) of an app as the average *gap* between it and all the apps it is linkable to. LE_a characterizes the difficulty in establishing linkability with a. $LE_a = 0$ means that to link information from app a and any random app it is linkable to, an adversary does not need additional information from a third app.

$$LE_a = \sum_{\substack{b \in LCC_a \\ b \neq a}} \frac{gap_{a,b}}{size(LCC_a) - 1}$$

LR and LE describe two orthogonal views of the DLG. In general, LR represents the quantity of links, describing the percentage of all installed apps that are linkable to a certain app, whereas LE characterizes the quality of links, describing the average amount of effort an adversary needs to make to link a certain app with other apps. In Fig. 3.1, $LR_A = 6/8$, $LR_H = 1/8$; $LE_A = \frac{0+0+0+1+1+2}{7-1} = 4/6$, $LE_H = 0$.

GLR and GLE Both LR and LE are defined for a single app, and we also need two similar definitions for the entire graph. So, we introduce *Global Linking Ratio* (GLR) and *Global Linking Effort* (GLE). GLR represents the probability of two randomly selected apps being linkable, while GLE represents the number of apps an adversary needs to control to link two random apps.

$$GLR = \sum_{a} \frac{LR_a}{size(DLG)}$$

$$GLE = \frac{1}{\sum_a size(LCC_a) - 1} \sum_{b} \sum_{\substack{c \in LCC_b \\ c \neq b}} gap_{b,c}$$

In graph theory, GLE is also known as the *Characteristic Path Length* (CPL) of a graph, which is widely used in Social Network Analysis (SNA) to characterize whether the network is easily negotiable or not.

3.3.3 Sources of Linkability

DLG maintains a dynamic view of app-level linkability by monitoring runtime behaviors of the apps. Specifically, it keeps track of apps' access to *device-specific* identifiers (IMEI, Android ID, MAC), *user-specific* identifiers (Phone Number, Accounts, Subscriber ID, ICC Serial Number), and *context-based* information (IP, Nearby APs, Location). It also monitors explicit IPC channels (Intent, Service Binding) and implicit IPC channel (Indirect RW, i.e., reading and writing the same file or database). This is not an exhaustive list but covers most standard and widely-used aggregating channels. Table 3.2 presents a list of all the contributing sources we consider and the details of each source will be elaborated in Section 3.4.

The criterion of two apps being linkable differs depending on the linkability source. For consistent identifiers that are obviously unique — Android ID, IMEI, Phone Number, MAC, Subscriber ID, Account, ICC Serial Number — two apps are linkable if they both accessed the same type of identifier. For pair-wise IPCs — intents, service bindings, and indirect RW — the two communicating parties involved are linkable. For implicit and fuzzy information, such as location, nearby APs, and IP, there are well-known ways to establish linkability as well. User-specific location clusters (Points of Interests, or PoIs) is already known to be able to uniquely identify a user [42, 56, 114]. Therefore, an adversary can link different apps by checking whether the location information they collected reveal the same PoIs. Here, the PoIs are extracted using a lightweight algorithm as used in [13, 36]. We select the top 2 PoIs as the linking standard, which typically correspond to home and work addresses. Similarly, the consistency and persistence of a user's PoIs are also reflected on its AP clusters and frequently-used IP addresses. This property allows us to establish linkability across apps using these fuzzy contextual information.

Category	*Type*	*Source*
OS-level Info.	Device	IMEI Android ID MAC
	Personal	Phone # Account Subscriber ID ICC Serial #
	Contextual	IP Nearby APs Location (PoIs)
IPC Channel	Explicit	Intent Service Binding
	Implicit	Indirect RW

Table 3.2: DLG considers the linkability introduced by 10 types of OS-level information and 3 IPC channels.

3.3.4 DLG: A Mobile OS Extension

DLG gives us the capability to construct cross-app linkability from runtime behaviors of the apps. Here, we introduce how it can be implemented as an extension to current mobile operating systems, using Android as an illustrative example. We also considered other implementation options, such as user-level interception (Aurasium [110]) or dynamic OS instrumentation (Xposed Framework [109]). The former is insecure since the extension resides in the attacker's address space and the latter is not comprehensive because it cannot handle the native code of an app. However, the developer can always implement a useful subset of DLG using one of these more deployable techniques.

Android Basics Android is a Linux-based mobile OS developed by Google. By default, each app is assigned a different Linux uid and lives in its own sandbox. Inter-Process Communications (IPCs) are provided across different sandboxes, based on the Binder protocol which is inherently a lightweight RPC (Remote Procedure Call)

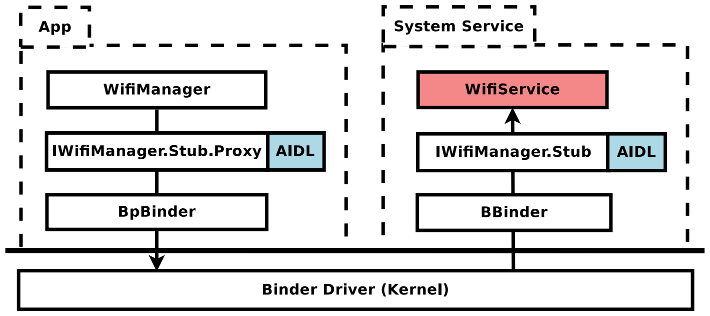


Figure 3.2: We instrument system services (red shaded region) to record which app accessed which identifier using Wi-Fi service as an example.

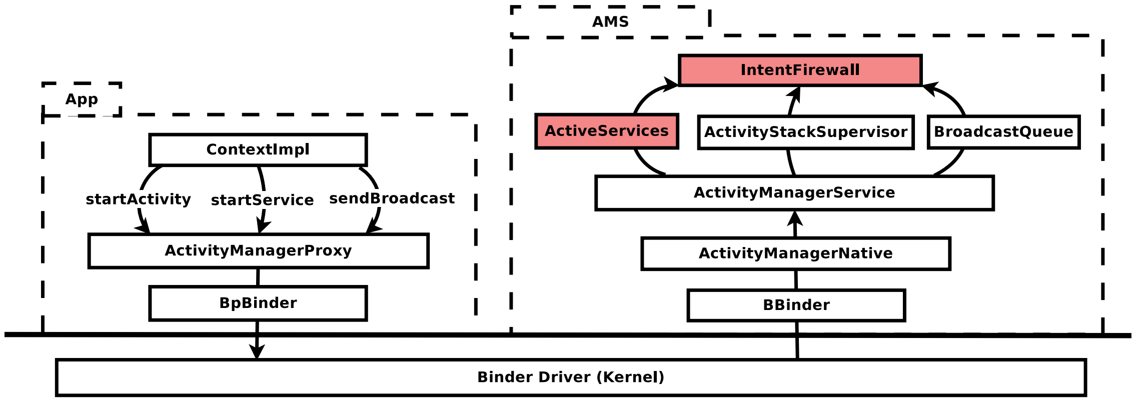


Figure 3.3: We extend the centralized intent filter implemented in the Android framework (`com.android.server.firewall.IntentFirewall`) to intercept all the Intents across apps.

mechanism. There are four different types of components in an Android app: Activity, Service, Content Provider, and Broadcast Receiver. Each component represents a different way to interact with the underlying system: Activity corresponds to a single screen supporting user interactions; Service runs in the background to perform long-running operations and processing; Content Provider is responsible for managing and querying of persistent data such as database; and Broadcast Receiver listens to system-wide broadcasts and filters those it is interested in. Next, we describe how we instrument the Android framework to monitor app's interactions with the system and each other via these components.

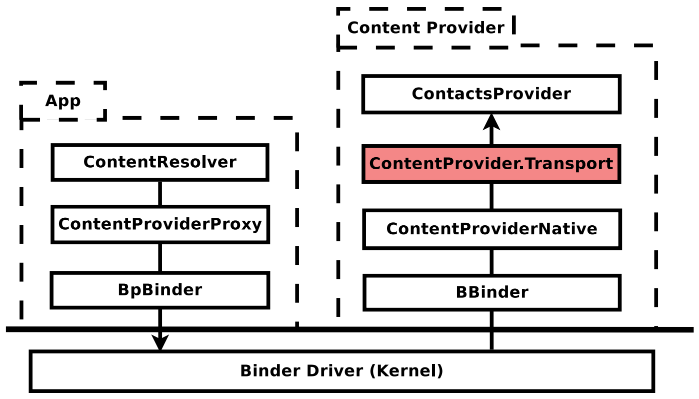


Figure 3.4: We instrument Content Provider (shaded region) to record which app accessed which database with what parameters.

Implementation Details In order to construct a DLG in Android, we need to track apps' access to various OS-level information as well as IPCs between apps. Next, we describe how we achieve this by instrumenting different components of the Android framework.

Apps access most identifying information, such as IMEI and MAC, by interacting with different system services. These system services are parts of the Android framework and have clear interfaces defined in AIDL (Android Interface Definition Language). By instrumenting the public functions in each service that return persistent identifiers, we can have a timestamped record of which app accessed what type of identifying information via which service. Fig. 3.2 gives a detailed view of where to instrument using the Wi-Fi service as an example.

On the other hand, apps access some identifying information, such as Android ID, by querying the content providers maintained by the system. Android framework has a universal choke point for all access to remote content providers — the server-side stub class `ContentProvider.Transport`. By instrumenting this class, we know which database (uri) an app is accessing and with what parameters and actions. Fig. 3.4 illustrates how an app accesses remote Content Provider and explains which part to modify in order to log the information we need.

Apps can launch IPCs explicitly, using Intents. Intent is an abstract description of an operation to be performed. It can either be sent to a specific target (app component), or broadcast to the entire system. The Android framework provides a centralized filter which enforces system-wide policies for all Intents. We choose to extend this filter (`com.android.server.firewall.IntentFirewall`) to record and intercept all Intent communications across apps (see Fig. 3.3). In addition to Intents, Android also allows an app to communicate explicitly with another app by binding to one of the services it exports. Once the binding is established, the two apps can communicate under a client-server model. We instrument `com.android.server.am.ActiveServices` in the Activity Manager to monitor all the attempts to establish service bindings across apps.

Apps can also conduct IPCs implicitly by exploiting shared persistent storage. For example, two apps can write and read the same file in the SD card to exchange identifying information. Therefore, we need to monitor read and write access to persistent storage. External storage in Android are wrapped by a FUSE (Filesystem in Userspace) daemon which enables user-level permission control. By modifying this daemon, we can track which app reads or writes which files (see Fig. 3.5). This allows us to implement a Read-Write monitor which captures implicit communications via reading a file which has previously been written by another app. Besides external storage, our Read-Write monitor also considers similar indirect communications via system Content Providers.

We described how to monitor all formal ways an app can interact with system components (Services, Content Providers) and other apps (Intents, service bindings, and indirect RW). This methodology is fundamental and can be extended to cover other potential linkability sources (beyond our list) as long as a clear definition is given. By placing hooks at the aforementioned locations in the system framework, we get all the information needed to construct a DLG. For our measurement study, we simply log and upload these statistics to a remote server for analysis. In our

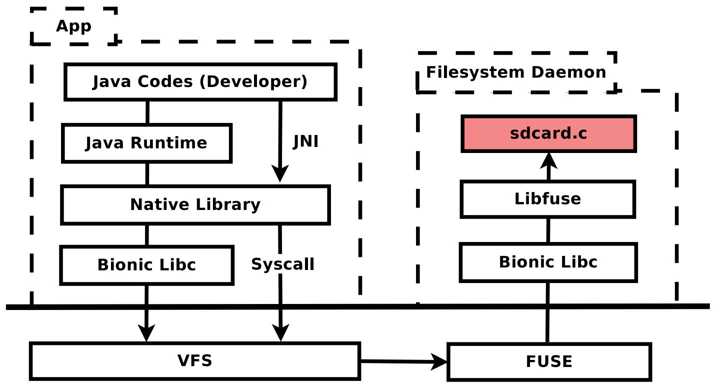


Figure 3.5: We customize the FUSE daemon under `/system/core/sdcard/sdcard.c` to intercept apps' access to shared external storage.

countermeasure solutions, these are used locally to derive dynamic defense decisions.

3.4 Linkability in Real World

In this section, we study app-level linkability in the real world. We first present an overview of linkability, showing the current threats we're facing. Then, we go through the linkability sources and analyze to what extent each of the sources is contributing to the linkability. Finally, we shed light on how these sources can be or have been exploited for reasons unrelated to app functionalities. This paves the way for us to develop a practical countermeasure.

3.4.1 Deployment and Settings

We prototyped DLG on Cyanogenmod 11 (based on Android 4.4.1) and installed the extended OS on 7 Samsung Galaxy IV devices and 6 Nexus V devices. We recruited 13 participants from the students and staff in our institution, spanning over 8 different academic departments. Of the 13 participants, 6 of the participants are females and 7 are males. Before using our experimental devices, 7 of them were Android users and 6 were iPhone users. Participants are asked to operate their devices normally without any extra requirement. They are given the option to temporarily

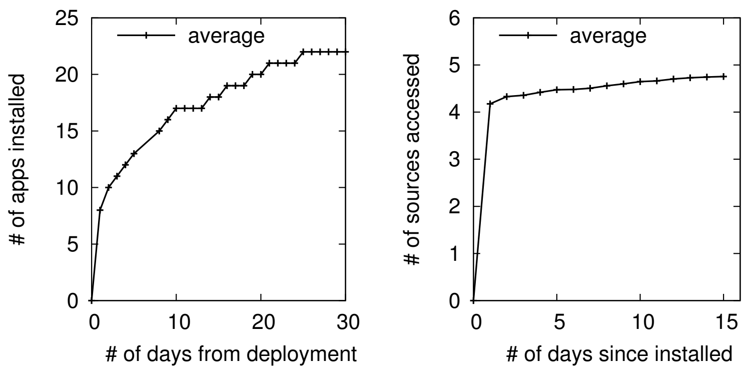


Figure 3.6: For an average user, more than 80% of the apps are installed in the first two weeks after deployment; each app accesses most of the linkability sources it's interested in during the first day of its installation.

turn off our extension if they want more privacy when performing certain tasks. Logs are uploaded once per hour when the device is connected to Wi-Fi. We exclude built-in system apps (since the mobile OS is assumed to be benign in our threat model) and consider only third-party apps that are installed by the users themselves. Note that our study is limited in its size and the results may not generalize.

3.4.2 Data and Findings

We observed a total of 215 unique apps during a 47-day period for 13 users. On average, each user installed 26 apps and each app accessed 4.8 different linkability sources. We noticed that more than 80% of the apps are installed within the first two weeks after deployment, and apps would access most of the linkability sources they are interested in during the first day of their installation (see Fig. 3.6). This suggests that a relative short-term (a few weeks) measurement would be enough to capture a representative view of the problem.

Overview: Our measurement indicates an alarming view of the threat: two random apps are linkable with a probability of 0.81, and an adversary only needs to control 2.2 apps (0.2 additional app), on average, to link them. This means that an adversary in the current ecosystem can aggregate information from most apps without additional

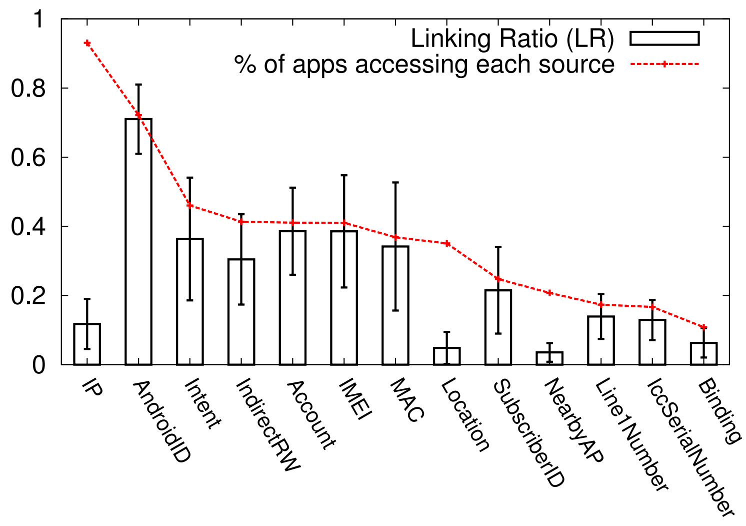


Figure 3.7: The percentage of apps accessing each source, and the linkability (LR) an app can get by exploiting each source.

efforts (i.e., controlling a third app). Specifically, we found that 86% of the apps a user installed on his device are directly linkable to the Facebook app, namely, his real identity. This means almost all the activities a user exhibited using mobile apps are identifiable, and can be linked to the real person.

Breakdown by Source: This vast linkability is contributed by various sources in the mobile ecosystem. Here, we report the percentage of apps accessing each source and the linkability (LR) an app can acquire by exploiting each source. The results are provided in Fig. 3.7. We observed that except for device identifiers, many other sources contributed to the linkability substantially. For example, an app can be linked to 39% of all installed apps (LR=0.39) using only account information, and 36% (LR=0.36) using only Intents. The linkability an app can get from a source is roughly equal to the percentage of apps that accessed that source, except for the case of contextual information: IP, Location and Nearby APs. This is because the contextual information an app collected does not always contain effectively identifying information. For example, Yelp is mostly used at infrequent locations to find nearby restaurants, but is rarely used at consistent PoIs, such as home or office. This renders

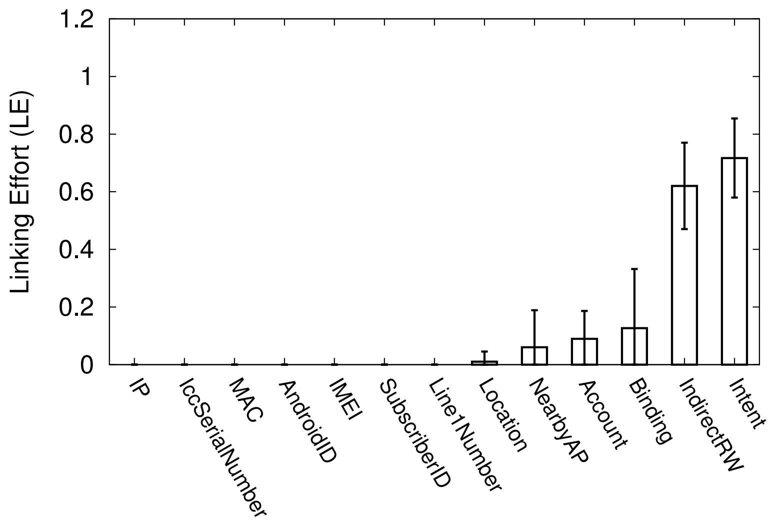


Figure 3.8: The (average) Linking Efforts (LE) of all the apps that are linkable due to a certain linkability source.

location information useless in establishing linkability with Yelp.

The effort required to aggregate two apps also differs for different linkability sources, as shown in Fig. 3.8. Device identifiers have *LE=0*, meaning that any two apps accessing the same device identifier can be directly aggregated without requiring control of an additional third app. Linking apps using IPC channels, such as Intents and Indirect RW, requires the adversary to control an average of 0.6 additional app as the connecting nodes. This indicates that, from an adversary's perspective, exploiting consistent identifiers is easier than building pair-wise associations.

Breakdown by Category: We group the linkability sources into four categories — device, personal, contextual, and IPC — and study the linkability contributed by each category (see Table 3.3). As expected, device-specific information introduces substantial linkability and allows the adversary to conduct across-app aggregation effortlessly. Surprisingly, the other three categories of linkability sources also introduce considerable linkability. In particular, only using fuzzy contextual information, an adversary can link more than 40% of the installed apps to Facebook, the user's real identity. This suggests the naive solution of anonymizing device ids is not enough,

Category	*GLR*	*GLE*	$LR_{Facebook}$
Device	0.52 (0.13)	0.03 (0.03)	0.68 (0.12)
Personal	0.30 (0.10)	0.30 (0.11)	0.54 (0.11)
Contextual	0.20 (0.13)	0.33 (0.20)	0.44 (0.25)
IPC	0.32 (0.13)	0.78 (0.06)	0.59 (0.15)

Table 3.3: Linkability contributed by different categories of sources.

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <long name="timestamp" value="1419049777098" />
    <long name="t2" value="1419049776889" />
    <string name="UTDID">VJT7MTV268gDACiZN6xEh8af</string>
    <string name="DID">356565055348652</string>
    <long name="S" value="1634341681" />
    <string name="SI">310260981039000</string>
    <string name="EI">356565055348652</string>
</map>
```

Figure 3.9: Real-world example of indirect RW: an app (fm.qingting.qradio) writes user identifiers to an xml file in SD card which was later read by three other apps. This file contains the IMEI (DID) and SubscriberID (SI) of the user.

and hence a comprehensive solution is needed to make a trade-off between app functionality and privacy.

3.4.3 Functional Analysis

Device identifiers (IMEI, Android ID, MAC) introduce vast amount of linkability. We manually went through 162 mobile apps that request these device-specific identifiers, but could rarely identify any explicit functionality that requires accessing the actual identifier. In fact, for the majority of these apps, their functionalities are device-independent, and therefore independent of device IDs. This indicates that device-specific identifier can be obfuscated across apps without noticeable loss of app functionality. The only requirement for device ID is that it should be unique to each device.

As to personal information (Account Number, Phone Number, Installed Apps, etc.), we also observed many unexpected accesses that resulted in unnecessary linkability. We found that many apps that request account information collected all user accounts even when they only needed one to function correctly; many apps request access to phone number even when it is unrelated to their app functionalities. Since the legitimacy of a request depends both on the user's functional needs and the specific app context, end-users should be prompted about the access and make the final decision.

The linkability introduced by contextual information (Location, Nearby AP) also requires better regulation. Many apps request permission for precise location, but not all of them actually need it to function properly. In many scenarios, apps only require coarse-grained location information and shouldn't reveal any identifying points of interest (PoIs). Nearby AP information, which is only expected to be used by Wi-Fi tools/managing apps, is also abused for other purposes. We noticed that many apps frequently collect Nearby AP information to build an internal mapping between locations and access points (APs). For example, we found that even if we turn off all system location services, WeChat (an instant messaging app) can still infer the user's location only with Nearby AP information. To reduce the linkability introduced by these unexpected usages, the users should have finer-grained control on when and how the contextual information can be used.

Moreover, we found that IPC channels can be exploited in various ways to establish linkability across apps. Apps can establish linkabililty using Intents, sharing and aggregating app-specific information. For instance, we observed that WeChat receives Intents from three different apps right after their installations, reporting their existence on the same device. Apps can also establish linkability with each other via service binding. For example, both AdMob and Facebook allow an app to bind to its service and exchanging the user identifier, completely bypassing the system permissions and controls. Apps can also establish linkabililty through Indirect RW, by writing and reading the same persistent file. Fig. 3.9 shows a real-world example: an app (fm.qingting.qradio) writes user identifiers to an xml file in the SD card which was later read by three other apps. The end-user should be promptly warned about these unexpected communications across apps to reduce unnecessary linkability.

3.5 LinkDroid: A Practical Countermeasure

Based on our observation and findings on linkability across real-world apps, we propose a practical countermeasure, `LinkDroid`, on top of DLG. We first introduce the basic design principle of `LinkDroid` and its three major privacy-enhancing features: *install-time obfuscation*, *runtime linkability monitoring*, and *unlinkable mode support*. We then evaluate the effectiveness of `LinkDroid` with the same set of participants as in our measurement study.

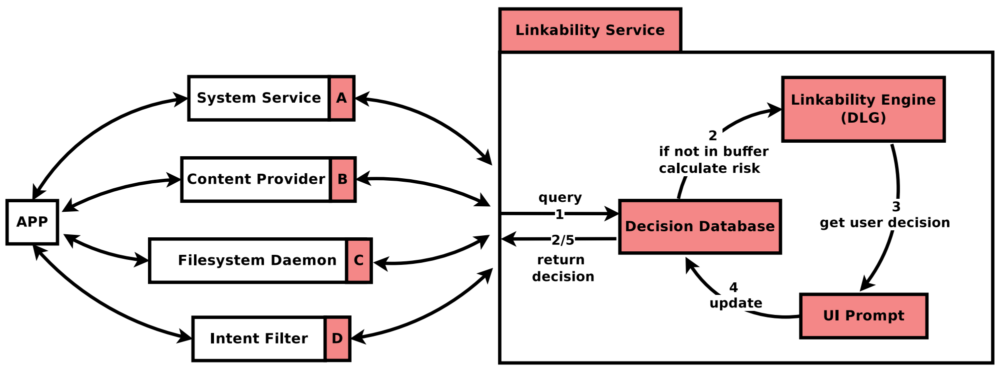


Figure 3.10: An overview of `LinkDroid`. Shaded areas (red) represent the parts we need to extend/add in Android. (We already explained how to extend A, B, C and D in Section 3.3.4.)

3.5.1 Design Overview

`LinkDroid` is designed with practicality in mind. Numerous extensions, paradigms and ecosystems have been proposed for mobile privacy, but access control (runtime for iOS and install-time for Android) is the only deployed mechanism. `LinkDroid` adds a new dimension to access control on smartphone devices. Unlike existing approaches that check if some app behavior poses direct privacy threats, `LinkDroid` warns users about how it implicitly builds the linkability across apps. This helps users reduce unnecessary links introduced by abusing OS-level information and IPCs, which happens frequently in reality as our measurement study indicated.

As shown in Fig. 3.10, `LinkDroid` provides runtime monitoring and mediation of linkability by

- monitoring and intercepting app behaviors that may introduce linkability (including interactions with various system services, content providers, shared external storage and other apps);
- querying a standalone linkability service to get the user's decision regarding this app behavior;
- prompting the user about the potential risk if the user has not yet made a decision, getting his decision and updating the linkability graph (DLG).

We have already described in Section 3.3.4 how to instrument the Android framework to build the monitoring components (corresponding to boxes A, B, C, D in Fig. 3.10). In this section, we focus on how the linkability service operates.

3.5.2 Install-Time Obfuscation

As mentioned earlier, app functionalities are largely independent of device identifiers. This allows us to obfuscate these identifiers and cut off many unnecessary edges in the DLG. In our case, the list of device identifiers includes IMEI, Android ID and MAC. Every time an app gets installed, the linkability service receives the app's uid and then generates a random mask code for it. The mask code together with the types of obfuscated device identifiers will be pushed into the decision database. This way, when an app a tries to fetch the device identifier of a certain type t, it will only get a hash of the real identifier salted with the app-specific mask code:

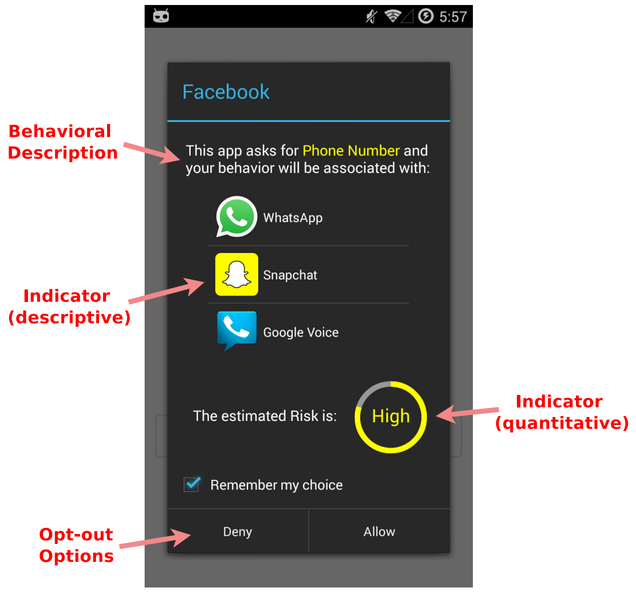
$$ID_t^a = hash(ID_t + mask_a).$$

Note that we do this at install-time instead of during each session because we still want to guarantee the relative consistency of the device identifiers within each app. Otherwise, it will let the app think the user is switching to a different device and trigger some security/verification mechanisms. The user can always cancel this default obfuscation in the privacy manager (Fig. 3.12) if he finds it necessary to reveal real device identifiers to certain apps.

3.5.3 Runtime Linkability Monitoring

Except for device-specific identifiers, obfuscating other sources of linkability is likely to interfere with the app functionalities. Whether there is a functional interference or not is highly user-specific and context-dependent. To make a useful trade-off, the user should be involved in this decision-making process. Here, `LinkDroid` provides just-in-time prompts before an edge creates in the DLG. Specifically, if the linkability service could not find an existing decision regarding some app behavior, it will issue the user a prompt, informing him: 1) what app behavior triggers the prompt; 2) what's the quantitative risk of allowing this behavior; and 3) what're the opt-out options. Fig. 3.11 gives an illustrative example of the UI of the prompt.

Description of App Behavior Before the user can make a decision, he first needs to know what app behavior triggers the prompt. Basically, we report two types of description: access to OS-level information and cross-app communications. To help the user understand the situation, we use a high-level descriptive language instead of the exact technical terms. For example, when an app tries to access Subscriber ID or


5:57

Facebook

This app asks for Phone Number and your behavior will be associated with:

WhatsApp

Snapchat

Google Voice

The estimated Risk is: High

Remember my choice

Deny

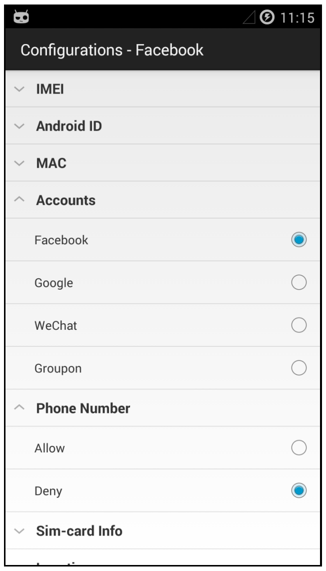
Allow

Figure 3.11: The UI prompt of `LinkDroid`'s runtime access control, consisting of a behavioral description, descriptive and quantitative risk indicators, and opt-out options.

IccSerialNumber, we report that "App X asks for sim-card information." When an app tries to send Intents to other apps, we report "App X tries to share content with App Y". During our experiments with real users (introduced later in the evaluation), 11 out of the 13 participants find these descriptions clear and informative.

Risk Indicator `LinkDroid` reports two types of risk indicators to users: one is descriptive and the other is quantitative. The descriptive indicator tells what apps will be directly linkable to an app if the user allows its current behavior. By 'directly linkable,' we mean without requiring a third app as the connecting nodes. The quantitative indicator, on the other hand, reflects the influence on the overall linkability of the running app, including those apps that are not directly linkable to it. Here, the overall linkability is reported as a combination of the linking ratio (LR) and linking effort (LE):

$$L_a = LR_a \times e^{-LE_a}.$$


11:15

Configurations - Facebook

IMEI

Android ID

MAC

Accounts

[✓ Facebook]

Google

WeChat

Groupon

Phone Number

Allow

[✓ Deny]

Sim-card Info

Figure 3.12: `LinkDroid` provides a centralized linkability manager. The user can review and modify all of his previous decisions regarding each app.

The quantitative risk indicator is defined as ΔL_a. A user will be warned of a larger risk if the total number of linkable apps significantly increases, or the average linking effort decreases substantially. We transform the quantitative risk linearly into a scale of 4 and report the risk as Low, Medium, High, and Severe.

Opt-out Options In each prompt, the user has at least two options: Allow or Deny. If the user chooses Deny, `LinkDroid` will obfuscate the information this app tries to get or shut down the communication channel this app requests. For some types of identifying information, such as Accounts and Location, we provide finer-grained trade-offs. For Location, the user can select from zip-code level (1km) or city-level (10km) precision; for Accounts, the user can choose which specific account he wants to share instead of exposing all his accounts. `LinkDroid` also allows the user to set up a VPN (Virtual Private Network) service to anonymize network identifiers. When the user switches from a cellular network to Wi-Fi, `LinkDroid` will automatically initialize the VPN service to hide the user's public IP. This may incur additional energy consumption and latency (see Section 3.5.5). All choices made by the user will be stored in the decision database for future reuse. We provide a centralized privacy manager such that the user can review and change all previously made decisions (see Fig. 3.12).

3.5.4 Unlinkable Mode

Once a link is established in DLG, it cannot be removed. This is because once a piece of identifying information is accessed or a communication channel is established, it can never be revoked. However, the user may sometimes want to perform privacy-preserving tasks which have no interference with the links that have already been introduced. For example, when the user wants to write an anonymous post in Reddit, he doesn't want it to be linkable with any of his previous posts as well as other apps. `LinkDroid` provides an unlinkable mode to meet such a need. The user can start an app in unlinkable mode by pressing its icon for long in the app launcher. A new uid as well as isolated storage will be allocated to this unlinkable app instance. By default, access to all OS-level identifying information and inter-app communications will be denied. This way, `LinkDroid` creates the illusion that this app has just been installed on a brand-new device. The unlinkable mode allows `LinkDroid` to provide finer-grained (session-level) control, unlinking only a certain set of app sessions.

3.5.5 Evaluation

We evaluate `LinkDroid` in terms of its overheads in usability and performance, as well as its effectiveness in reducing linkability. We replay the traces of the 13 participants of our measurement study (see Section 3.4), prompt them about the privacy threat and ask for their decisions. This gives us the exact picture of the same set of users using `LinkDroid` during the same period of time. We instruct the user to make a decision in the most conservative way: the user will Deny a request only when he believes the prompted app behavior is not applicable to any useful scenario;

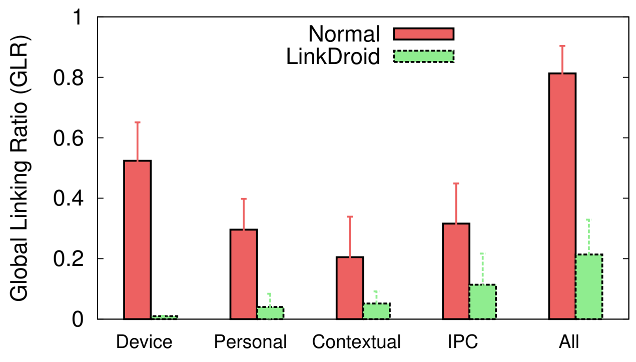


Figure 3.13: The Global Linking Ratio (GLR) of different categories of sources before and after using `LinkDroid`.

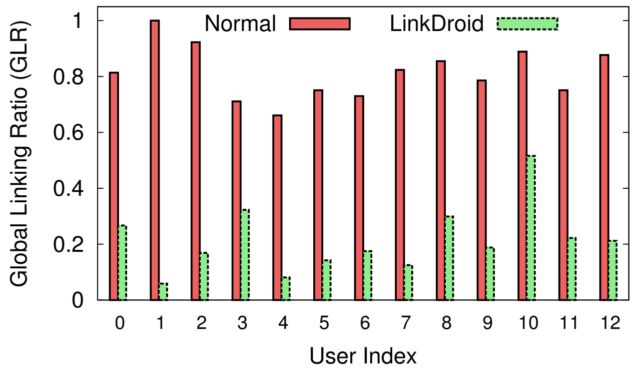


Figure 3.14: The Global Linking Ratio (GLR) of different users before and after using `LinkDroid`.

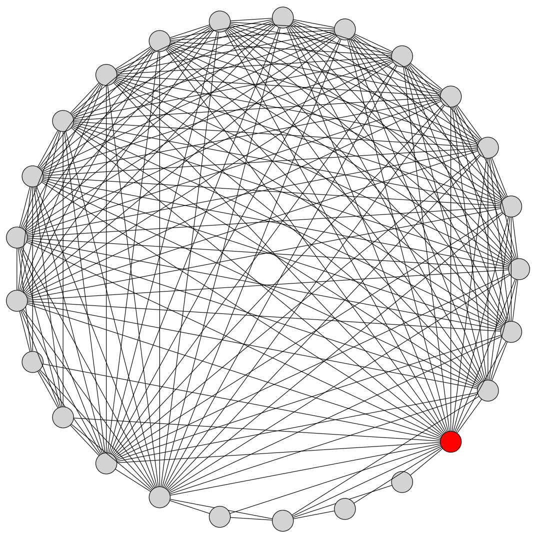
otherwise, he will Accept the request.

The overhead of `LinkDroid` mainly comes from two parts: the usability burden of dealing with UI prompts and the performance degradation of querying the linkability service. Our experimental results show that, on average, each user was prompted only 1.06 times per day during the 47-day period. The performance degradation introduced by the linkability service is also marginal. It only occurs when apps access certain OS-level information or conduct cross-app IPCs. These sensitive operations happened rather infrequently — once every 12.7 seconds during our experiments. These results suggest that `LinkDroid` has limited impact on system performance and usability.

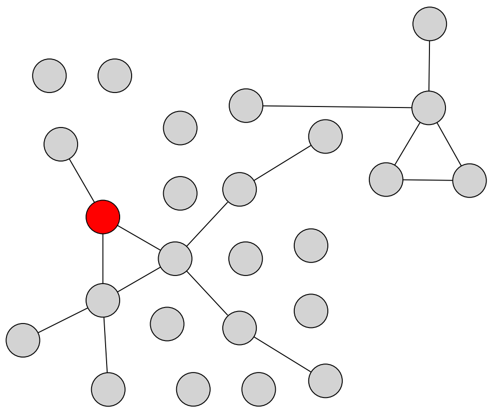
We found that after applying `LinkDroid`, the Global Linking Ratio (GLR) dropped from 81% to 21%. Fig. 3.13 shows the breakdown of linkability drop in different categories of sources. The majority of the remaining linkability comes from inter-app communications, most of which are genuine from the user's perspective. Not only fewer apps are linkable, `LinkDroid` also makes it harder for an adversary to aggregate information from two linkable apps. The Global Linking Effort (GLE) increases significantly after applying `LinkDroid`: from 0.22 to 0.68. Specifically, the percentage of apps that are directly linkable to Facebook dropped from 86% to 18%. Fig. 3.15 gives an illustrative example of how DLG changes after applying `LinkDroid`. We also noticed that that the effectiveness of `LinkDroid` differs across users, as shown in Fig. 3.14. In general, `LinkDroid` is more effective for the users who have diverse mobility patterns, are cautious about sharing information across apps and/or maintain different accounts for different services.

`LinkDroid` takes VPN as a plug-in solution to obfuscate network identifiers. The potential drawback of using VPN is its influence on device energy consumption and network latency. We measured the device energy consumption of using VPN on a Samsung Galaxy 4 device, with Monsoon Power Monitor. Specifically, we tested two network-intensive workloads: online videos and browsing. We observed a 5% increase in energy consumption for the first workload, and no observable difference for the second. To measure the network latency, we measured the ping time (average of 10 trials) to Alexa Top 20 domains and found a 13% increase (17ms). These results indicate that the overhead of using VPN on smartphone device is noticeable but not significant. Seven of 13 participants in our evaluation were willing to use VPN services to achieve better privacy.

We interviewed the 13 participants after the experiments. Questions are designed on a scale of 1 to 5 and a score of 4 or higher is regarded as "agree." Eleven of the participants find the UI prompt informative and clear and nine are willing to



(a)



(b)

Figure 3.15: DLG of a representative user before (a) and after (b) applying `LinkDroid`. Red circle represents the Facebook app.

use `LinkDroid` on a daily basis to inform them about the risk and provide opt-out options. However, these responses might not be representative due to the limited size and diversity of the participants. We also noticed that users care a lot about the linkability of sensitive apps, such as Snapchat and Facebook. Some participants clearly state that they do not want any app to be associated with the Facebook app, except for very necessary occasions. This also supports the rationale behind the design of `LinkDroid`'s unlinkable mode.

3.6 Related Work

There have been other proposals [29, 60] which also address the privacy threats of information aggregation by mobile apps. They shift the responsibility of information personalization and aggregation from mobile apps to the mobile OS or trusted cloud providers, requiring re-development of mobile apps and extensive modifications on the entire mobile ecosystem. In contrast, `LinkDroid` is a client-side solution which is compatible with existing ecosystem — it focuses on characterizing the threat in current mobile ecosystem and making a practical trade-off, instead of proposing new computation (advertising) paradigm.

Existing studies investigated linkability under several domain-specific scenarios. Arvind *et al.* [79] showed that a user's profile in Netflix can be effectively linked to his in IMDB, using long-tailed (unpopular) movies. Sebastian *et al.* [58] described how to link the profiles of the same user in different social networks using friends topologies. This type of linkability is restricted to a small scope, and may only exist across different apps in the same domain. Here, we focus on the linkability that are domain-independent and ubiquitous to all apps, regardless of the type and semantics of each app.

The capability of advertising agency on conducting profiling and aggregation has been extensively studied [45, 98]. Various countermeasures have been proposed, such as enforcing finer-grained isolation between ad library and the app [86, 95], or adopting a privacy-preserving advertising paradigm [10]. However, unlike `LinkDroid`, they only consider a very specific and restricted scenario — advertising library — which involves few functional trade-offs. `LinkDroid`, instead, introduces a general linkability model, considers various sources of linkability and suits a diverse set of adversaries.

There have also been numerous studies on information access control on smartphone [18, 31, 33, 49, 53, 81, 101]. Many of these studies have already proposed to provide apps with fake identifiers and other types of sensitive information [49, 81, 109]. These studies focus on the explicit privacy concern of accessing and leaking sensitive user information, by malicious mobile apps or third-party libraries. Our work addresses information access control from a very different perspective, investigating the implicit linkability introduced by accessing various OS-level information and IPC channels.

Many modern browsers provide a private (incognito) mode. These are used to defend against local attackers, such as users sharing the same computer, from stealing cookies or browse history from each other [3]. This is inherently different from `LinkDroid`'s *unlinkable mode* which targets unregulated aggregation by remote attackers.

3.7 Discussion

In this chapter, we proposed a new metric, *linkability*, to quantify the ability of different apps to link and aggregate their usage behaviors. This metric, albeit useful, is only a coarse upper-bound of the actual privacy threat, especially in the case of IPCs. Communication between two apps does not necessarily mean that they have conducted, or are capable of conducting, information aggregation. However, deciding on the actual intention of each IPC is by itself a difficult task. It requires an automatic and extensible way of conducting semantic introspection on IPCs, and is a challenging research problem on its own.

`LinkDroid` aims to reduce the linkability introduced covertly without the user's consent or knowledge — it couldn't and doesn't try to eliminate the linkability explicitly introduced by users. For example, a user may post photos of himself or exhibit very identifiable purchasing behavior in two different apps, thus establishing linkability. This type of linkability is app-specific, domain-dependent and beyond the control of `LinkDroid`. Identifiability or linkability of these domain-specific usage behaviors are of particular interest to other areas, such as anonymous payment [106], anonymous query processing [77] and data anonymization techniques.

The list of identifying information we considered in this work is well-formatted and widely-used. These ubiquitous identifiers contribute the most to information aggregation, since they are persistent and consistent across different apps. We didn't consider some uncommon identifiers, such as walking patterns and microphone signatures, because we haven't yet observed any real-world adoption of these techniques by commercial apps. However, `LinkDroid` can easily include other types of identifying information, as long as a clear definition is given.

DLG introduces another dimension — linkability — to privacy protection on mobile OS and has some other potential usages. For example, when the user wants to perform a certain task in Android and has multiple optional apps, the OS can recommend him to choose the app which is the least linkable with others. We also noticed some interesting side-effect of `LinkDroid`'s unlinkable mode. Since unlinkable mode allows users to enjoy finer-grained (session-level) unlinkability, it can be used to stop a certain app from continuously identifying a user. This can be exploited to infringe the benefits of app developers in the case of copyright protection, etc. For example, NYTimes only allows an unregistered user to read up to 10 articles every month. However, by restarting the app in unlinkable mode in each session, a user can stop NYTimes from linking himself across different sessions and bypass this quota restriction.

3.8 Conclusion

In this chapter, we addressed the privacy threat of unregulated aggregation from a new perspective by monitoring, characterizing and reducing the underlying linkability across apps. This allows us to measure the potential threat of unregulated aggregation during runtime and promptly warn users of the associated risks. We observed how real-world apps abuse OS-level information and IPCs to establish linkability, and proposed a practical countermeasure, `LinkDroid`. It provides runtime monitoring and mediation of linkability across apps, introducing a new dimension to privacy protection on mobile device. Our evaluation on real users has shown that `LinkDroid` is effective in reducing the linkability across apps and only incurs marginal overheads.

CHAPTER IV

Enabling Unlinkability of Mobile Apps in User-level

4.1 Introduction

In the last chapter, we systematically studied the issue of unregulated aggregation of mobile app usages, when a curious party covertly links and aggregates a user's behavioral information collected from independent sources — across sessions and apps — without his consent or knowledge. The severity and prevalence of this threat are rooted at the nonexistence of unlinkability in the smartphone ecosystem. By exploiting various levels of consistency provided by device identifiers, software cookies, IPs, local and external storages, an adversary can easily correlate app usages of the same user and aggregate supposed-to-be 'isolated islands of information' into a comprehensive user profile, irrespective of the user's choice and (dis)approval. However, from the user's perspective, only app usages that are functionally-dependent should be linkable. For example, for GTalk, app usages under the same login should be linkable to provide a consistent messaging service. For Angry Birds, usage of the same app should be linkable to allow the user to resume from where he stopped. In contrast, for most query-like apps, such as Bing and Wikipedia, which neither enforce an explicit login nor require consistent long-term 'memories', app usages should be globally unlinkable by default.

Moreover, existing mobile OS vendors may be unlikely to adopt changes/extensions that can improve the privacy of users because the entire mobile ecosystem is fueled by the abundance of user information. To bridge this gap in a timely manner, we propose `Mask`, the first user-level solution that allows the user to negotiate to what extent his behavior can be linked and aggregated. Specifically, `Mask` introduces a set of *private execution modes* that allow the users to maintain multiple isolated profiles for each app. Each app profile can be temporary, being recycled after each session, or enduring, persists across multiple sessions. Upon invocation of each app, a user can apply one of the following modes to the current *app session* (from the start to termination of the app), according to his usage scenario. If he wants to:

- Use this app while logged in, choose the identifiable mode. All app usages under the same login are now linkable and the app delivers uncompromised personal services while disallowing aggregation across unrelated apps.
- Keep states or use the states saved before, apply the pseudonymous mode. In this mode, a user can maintain multiple profiles and only app usages in the same profile are linkable.
- Execute this app without leaving any trace, apply the anonymous mode. Each session is treated as independent and app usages are confined within the current session.

All user behaviors originate from the mobile apps, either directly or indirectly. By enabling the aforementioned private execution modes which isolate app usages at this very source, `Mask` provides a client-side solution without requiring any change to the existing ecosystem. We have implemented `Mask` on Android at user level, without requiring any modifications on the Android framework. This is achieved by enforcing a lightweight user-level sandbox that creates an isolated runtime environment with stripped account information, anonymized device IDs & software cookies, and isolated persistent storage. Our solution is able to bring privacy benefits to more than 70% of the apps and incurs negligible overhead in the app's runtime performance. Our user study on 27 users find that more than 60% of them find it useful to maintain multiple isolated profiles for mobile apps and 11 of them are willing to use this feature on a daily basis.

The rest of this chapter is organized as follows. The next section introduces the background of unregulated aggregation of mobile app usages with a focus on the practical challenges encountered. Section 4.3 presents a high-level design of `Mask`, while Section 4.4 describes our user-level implementation on Android as well as some evaluation results. Section 4.5 covers the related work, and finally Section 4.6 concludes the chapter.

4.2 Background & Challenges

4.2.1 Unregulated Aggregation

In current mobile ecosystems, an interested/curious party can covertly link and aggregate app usages of the same user over time, without his consent or knowledge, which we call *unregulated aggregation* of app usages. Here, we describe three major adversaries — mobile apps, A&A agencies, and network sniffers — and show how they aggregate app usages in practice.

4.2.1.1 Smartphone Applications

Smartphone apps aggregate users' app usages mainly for personalization. By tracking app usages over time and feeding them to domain-specific mining/learning algorithms, smartphone apps can deliver contents tailored to each user. Even if a user doesn't give an explicit consent (by logging in), apps can still identify and aggregate usages of the same user. In fact, if only for the purpose of user tracking, mobile apps have options far easier and simpler than enforcing login. Specifically, a smartphone app can use device IDs or system IDs, such as IMEI and Android ID, as a consistent user identity to aggregate his app usages remotely on the server, or exploit the consistent & persistent storage on the device and achieve the same goal locally.

4.2.1.2 Advertising & Analytics Agencies

To enable targeted advertising, A&A agencies are also interested in aggregating personal interests and demographics disclosed in app usages. Specifically, app developers include clients of these ad agencies—ad libraries—into their apps and proactively feed sensitive information requested by these libraries [98]. Moreover, since an ad library shares the same permission with its host app, it can also access and collect private information on its own. To identify and aggregate information of the same user, third-party libraries embed user identifiers into the traffic they send to the back-end servers. Such a user identifier can be the hash value of a device/system ID or a local cookie. These A&A agencies can be more dangerous than smartphone apps as they can aggregate usage behaviors across multiple apps carrying the same library.

4.2.1.3 Network Sniffers

Unlike the aforementioned parties, a network sniffer cannot collect information directly from the user's device on its own and can only extract information from raw network traffic. Moreover, from the sniffer's perspective, the network traffic can be really messy: some are directly marked with the actual device ID, some are tagged with hashed ones, some only embed app-specific ID (such as a craigslist user ID) while some others are encrypted and completely useless. However, as MOSAIC [108]

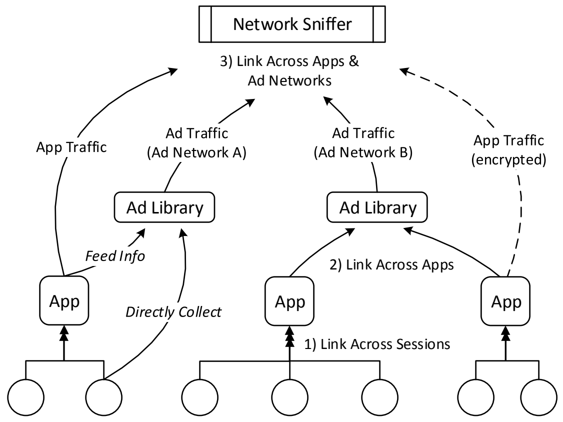


Figure 4.1: How usages in different *app sessions* (circles in the figure) are collected, aggregated and propagated by smartphone apps, A&A agencies and network sniffers.

shows, by exploiting the relative consistency of IP, it can associate different IDs that represent the same user. This way, even the traffic marked with different and seemingly unrelated user IDs can be aggregated. As publicized recently, a similar technique is used by government agencies (e.g., US NSA and GCHQ) for public surveillance.

In summary, these parties have an increasing scope of information collection and aggregation, and decreasing control on the client side (mobile device). Besides, they're not independent parties, but operate more like subordinates of an integrated adversary. Fig. 4.1 illustrates the information flow among these parties.

4.2.2 Practical Challenges

The following practical challenges need to be addressed when developing a solution.

4.2.2.1 Intermingled Interests of Different Parties

Free apps dominate mobile app stores with 91% of the overall downloads [1], and app developers include ad libraries to monetize these free apps. Therefore, smartphone apps share the same financial interest with A&A agencies, and should thus not be trusted. In fact, smartphone apps may deliberately collude with A&A agencies and feed them user demographics, such as age and gender, which ad libraries wouldn't be able to know on their own. On the other hand, OS vendors, whose popularity highly depends on the activeness of app developers, are reluctant to add privacy-enhancing features that may undermine the app developers' financial interests. Therefore, any defense that requires extensive OS-level modifications/cooperations will be impractical and not deployable.

4.2.2.2 Overpopulated User Identifiers

Unlike the web case where users are usually identified and tracked using cookies, smartphone apps have much more choices. Exploiting the consistency provided by numerous device and system IDs (some of which do not even require a permission to access), they can track app users both consistently and persistently. Moreover, since apps have arbitrary control over their persistent storage, they can perform local aggregation of users' information which doesn't even require any type of ID.

4.2.2.3 Trade-off between Functionality & Privacy

Enhancement of privacy often implies sacrifice of functionality. Similarly, opt-out unregulated aggregation, while providing a better privacy guarantee, also impairs personalized user experience. This trade-off, instead of being context-invariant, is subject to an app's nature as well as the user's preference. Thus, one must make a useful and adjustable trade-off between privacy and functionality.

These fundamental issues greatly reduce the set of tools and techniques a practical solution can use, rendering most existing proposals ineffective in practical settings (see Section 5 for the state-of-art).

4.3 MASK: A Client-side Design

4.3.1 Basic Design Idea

The basic idea behind `Mask`'s design is to allow only those app usages that are functionally dependent on each other to be linkable while keeping others unlinkable by default. This is achieved by introducing a set of *private execution modes* through which app users can provide explicit consent, on whether and within which scope the app usages in current *session* can be aggregated. The private execution modes are introduced based on our observation of how apps are actually used. Specifically, we first classify app-usage scenarios according to the levels of linkability required by app functionality and then introduce different private execution modes according to these app usage patterns.

4.3.2 App Usage Patterns

In `Mask`, the basic unit of a user's app usage is *session*, which represents a series of continuous active interactions between the user and an app to achieve a specific function. On Android, this typically corresponds to the activities between the invocations of function calls `onCreate` and `onDestroy`. The duration of a session is relatively short. Therefore, personal information in a single session can be very limited, and hence, different parties are devoted to linking and aggregating different sessions of the same user.

`Mask` classifies app usage patterns depending on whether and to what extent app usages in different sessions should be linkable. The three app usage patterns introduced below—*stateless*, *durative* and *exclusive*—characterize app's increasing need on the level of consistency in its runtime environment.

4.3.2.1 Stateless Pattern

the user's activity in one session does not depend strongly on the states of, and information from other sessions. By 'not strongly', we mean the the app is able to deliver its main functionality without any information from previous sessions, possibly at the expense of reducing optional personalized features. A wide spectrum of apps fit this pattern, including query-like apps such as Wikipedia and Yelp, apps from most news media such as NYTimes and CNN, and simple games such as Doodle Jumps and Flappy Bird.

4.3.2.2 Durative Pattern

the user requires persistent states and long-term 'memories' of an app to perform his current activities, but does not need to reveal his real-world identity. This pattern fits note-keeping apps, music player, books & magazines, complex games with a storyline or levels that need to be unlocked (such as Angry Birds), and etc.

4.3.2.3 Exclusive Pattern

the user must execute the app with an explicit identity, such as a user ID or account, and is willing to take the accompanied privacy risk. It covers most of social apps, such as Facebook and Twitter, as well as communication apps including WhatsApp, Yahoo Messenger, etc. If an app fits this pattern, the corresponding usages can and should be linkable across all the sessions that share the same account.

Note that the different users might apply different patterns when using the same app. This is subject to the preference of each specific user.

4.3.3 Aligning Usage Pattern with Privacy

Let's first discuss the default scenario in a contemporary mobile OS, using Android as an example. An app can track a user via device IDs such as IMEI or MAC address,

Table 4.1: The gap of linkability: Expectation vs. Reality

Scenario	*Linkable*	
	Across Sessions	*Across Apps*
stateless	single	single
durative	some - all	single
exclusive	all	single - some
default (reality)	all	all

which typically require permission, or via system IDs such as SERIAL number and android ID, which do not require any permission at all. If an app or an ad library wants, it can always export a cookie to its local storage, or more persistently, to external storage. These persistent anchors in the app's runtime allow an adversary to link and aggregate usage across all apps and sessions. However, from the user's perspective, this linkability is far too strong for most app functionalities.

For apps executed *statelessly*, linkability is not needed even in the weakest form since each session is inherently independent. For apps executed *duratively*, linkability is only needed across (some, not necessarily all) sessions of the same app—for example, a user may wish to maintain two separate instances for the same gaming app. Even for apps executed *exclusively* there are additional privacy issues. When a user executes an app with login, its app usages should only be linkable within the app, or at most across apps using the same login — instead of across all sessions and apps. Table 4.1 summarizes this gap between expectation and reality.

4.3.4 Private Execution Modes

Having understood the user's requirement on linkability, we present an intuitive way for the user to give explicit consent. Specifically, `Mask` introduces a set of *private execution modes*. Whenever a user starts an app, he can choose which mode to apply on the current session, implicitly specifying whether and within which scope his app usages can be aggregated.

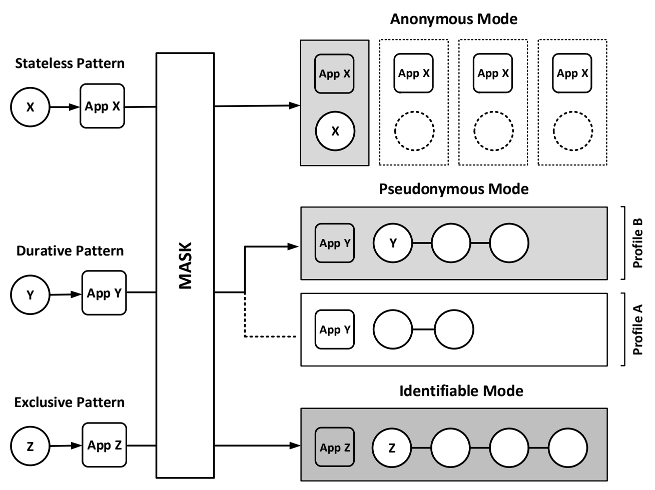


Figure 4.2: `Mask` provides different *private execution modes* for each usage pattern we classified and only app *sessions* that are functionally dependent on each other are linkable.

`Mask` provides three types of private execution modes—identifiable, pseudonymous and anonymous—which are mapped to the three usage patterns we defined earlier and provide increasing levels of unlinkability of the user's app usages. An ordinary user can make this decision by following a few simple rules, without any domain-specific knowledge. Specifically, when an app starts execution and the user wants to:

- Use this app while logged in, he should choose the *identifiable mode*. Assuming all app usages under the same login are linkable, `Mask` allows an app to deliver uncompromised personal services while disallowing aggregation across unrelated apps.
- Keep states or use the states saved before, he should apply the *pseudonymous mode*. In this mode, a user can maintain multiple context-based profiles and only app usages in the same profile are linkable.
- Execute this app without leaving any trace, he should apply the *anonymous mode*. Each session is treated as independent and app usages are confined within the current session.

Fig. 4.2 presents an overview of `Mask`'s design. Note that our design is not restricted to any specific mobile OS or platform since its rationality is rooted in the general notions of app usage patterns. Different choices of `Mask`'s implementation only reflect emphasis on different aspects, such as performance, robustness or practicality.

4.4 Implementation & Evaluation

The principle that drives every decision in our implementation is deployability without any unrealistic dependencies or assumptions on other parties, such as platform-level support or collaboration with A&A agencies. This is important because (1) the privacy threat under consideration is prevalent and needs to be dealt with urgently and users should be given a choice to opt out right away; and (2) as we discussed earlier, there exist intermingled benefits among different parties in the current mobile app ecosystem and counting on any of them may degrade the practicality of a solution. Guided by this principle, we developed a client-side prototype of `Mask` on Android (4.1.1) at the user level. Next, we provide the technical details and challenges of our user-level implementation.

4.4.1 User-Level Sandbox

To enable the aforementioned private execution modes, we need to provide an isolated runtime environment. Since practicality is priority in our implementation and users are less likely to use a custom ROM or root their device solely for privacy protection, we need a user-level sandbox implementation.

As proposed by the system communities [100], the dynamic linking process can be exploited to support program customization. Any dynamically linked executable keeps a mapping between external function symbols and the corresponding memory addresses, known as the *global offset table* (GOT). By rewriting entries in the GOT, access to any external function symbols can be redirected to a user-specified function. This makes it possible to intercept library calls in user-level and deliver security and privacy features [110].

We adopt this user-level technique to achieve two goals: intercepting inter-process communications (IPCs) between system services and apps to strip personal and device identifying information that enable aggregation at the server side; and provide an isolated per-sandbox storage to break local (on-device) aggregation.

4.4.1.1 IPC (Intent) Interception

IPC is the only supported mechanism in Android that allows an app to interact with other processes and exchange information. Any explicit communication, using *Intents*, or implicit ones, such as getting information from system services using high-level APIs, are supported by this IPC mechanism. To strip personal and device identifying information an app could get, we need to be able to intercept, understand and modify any IPCs between this app and other parties. This brings some technical challenges and requires a good understanding of how IPC works in Android.

In Android, the design of IPC, `Binder`, is conceptually a lightweight RPC mechanism which allows one process to invoke routines in another process. It consists of two components: the shared library `libbinder.so` in user space and the Binder driver in kernel space. They communicate with each other according to the Binder protocol via the bionic libc call *ioctl*. All high-level objects such as *Intents* are packed into a container object (Parcel) and then sent through the binder protocol as a byte array. By intercepting the *ioctl* function call in `libbinder.so`, we can exercise arbitrary user-level control. Specifically, we overwrite the GOT in `libbinder.so` and redirect *ioctl* to a wrapper function. This wrapper allows us to intercept both incoming and outgoing communications, both are indispensable to achieve our goal. Intercepting outgoing traffic lets us know what request this app sends while intercepting the incoming traffic allows us to change the results returned. Fig. 4.3 summarizes this process.

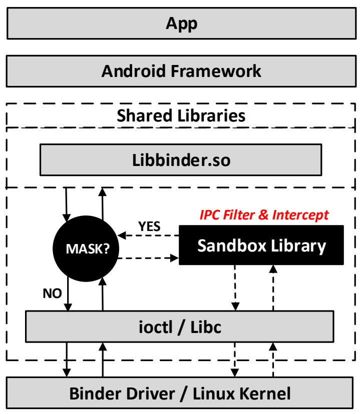


Figure 4.3: How to achieve Binder IPC interception.

On top of this interception mechanism, we can impose control over any intra- or inter-app communications as well as the app's interactions with system components. Here, we focus on the latter because in Android, identifying information is centrally managed by system services. Table 4.2 summarizes the list of identifiers `Mask` anonymizes. It contains the most commonly-used IDs but may not be a complete list of all potential identifying information. However, the associated technical underpinning is general enough to cover other identifiers, if necessary. We also exclude quasi-identifiers, such as IP or location, because compared to the the explicit identifiers addressed in `Mask`, they are far less consistent and reliable [11, 108], especially in the mobile context. Moreover, there are already independent lines of research on these subjects, such as IP anonymization and location anonymity.

4.4.1.2 Persistent Storage Isolation

Isolating persistent storage for each sandbox is necessary because it prevents local aggregation of the user's app usages, and also break the consistency of software cookies. Android provides the following options for persistent storage: Shared Preferences, Internal Storage, External Storage and SQLite Databases.

All of these storage options are built upon file system primitives provided by `Bionic Libc`, such as *open*, *stat*, *mkdir*, *chmod*, etc. By intercepting these primitives

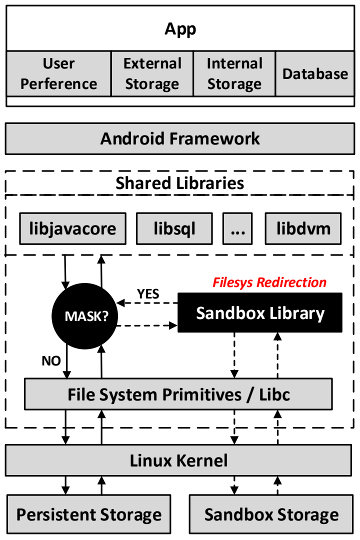


Figure 4.4: How to achieve persistent storage redirection/isolation.

Table 4.2: List of commonly-used identifiers `Mask` anonymizes

ID	*System Service*	*Permission*
IMEI/MEID	iphonesubinfo	READ_PHONE_STATE
SUBSCRIBER ID	iphonesubinfo	READ_PHONE_STATE
PHONE NUMBER	iphonesubinfo	READ_PHONE_STATE
MAC ADDR	wifi	ACCESS_WIFI_STATE
ACCOUNTS	accounts	GET_ACCOUNTS
ANDROID ID	settings	NONE
SERIAL	settings*	NONE

and modifying the corresponding input parameters, we can exercise arbitrary control over the app's interactions with the file system. Specifically, we create shadow directories which resemble the initial states of the app for each sandbox upon its creation, and then redirect all upcoming file system operations from the app-specific directories to the corresponding shadow directories. Figure. 4.4 illustrates this process.

4.4.2 Sandbox Manager

So far, we have introduced how a user-level sandbox is implemented to provide an isolated runtime environment. Next, we describe how these sandboxes are created, executed and destroyed to deliver the *private execution modes* in `Mask`.

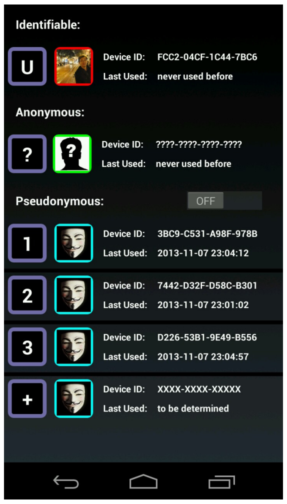
4.4.2.1 Sandbox Management

The lifecycle of each sandbox is centrally controlled by a sandbox manager. The sandbox manager maintains a meta file for each sandbox, which contains sandbox-specific parameters: paths to the designated shadow directories, anonymized values of the persistent identifiers. When a sandbox is created, the manager generates a sandbox-specific meta file and allocates it shadow directories both in the local storage and the SD card. Then, any resources required for the initial states of this app will be copied into the shadow directories. The sandbox is then ready to start. When a sandbox is executed, the manager will initialize the runtime with information stored in the sandbox's meta file and activate the library-call interception mechanism. When a sandbox is destroyed, the sandbox manager first clears the local storage designated for the sandbox and then deletes the corresponding meta file.

Whenever an app is executed in anonymous mode, a new sandbox is created and applied; the sandbox will be immediately destroyed after the current session terminates. Note that, in Android, the termination of a session (`onDestroy`) is automatically controlled/optimized by the system. The user can force the current session to terminate by removing the app from the recent apps list. If an app is executed in pseudonymous mode, the user can choose to reuse an existing sandbox or create a new sandbox; a sandbox will only be destroyed when a user explicitly specifies it. If an app is executed in identifiable mode, a designated sandbox gets initialized; the user always runs the same sandbox.

4.4.2.2 Multi-process Support

For an app with only one process, its execution in a sandbox is simple; but for apps with multiple processes, it can be complicated. Since what we implemented is a per-process sandbox and Android allows an app to host multiple processes, an app will crash if different processes are executed with inconsistent runtime environment.


Identifiable:

U — Device ID: FCC2-04CF-1C44-7BC6
Last Used: never used before

Anonymous:

? — Device ID: ????-????-????-????
Last Used: never used before

Pseudonymous: OFF

1 — Device ID: 3BC9-C531-A98F-978B
Last Used: 2013-11-07 23:04:12

2 — Device ID: 7442-D32F-D58C-B301
Last Used: 2013-11-07 23:01:02

3 — Device ID: D226-53B1-9E49-B556
Last Used: 2013-11-07 23:04:57

+ — Device ID: XXXX-XXXX-XXXXX
Last Used: to be determined

Figure 4.5: The UI design of `Mask`.

Therefore, we equip our sandbox with multi-process support. Each time an app process starts, the sandbox manager will first tell whether a sandbox is already created for this app (by maintaining a lock file in this app's local storage). If so, the new process will join the existing sandbox and share the same runtime environment.

4.4.2.3 UI Design

The logic of sandbox manager is hidden behind an intuitive UI. `Mask`'s UI is displayed right before the launcher activity of an app starts, and is executed as an independent process isolated from all other components of the app. It offers a nice and intuitive way for end-users to manage profiles without revealing too much details. As shown in Fig. 4.5, `Mask` provides three *execution modes*—identifiable, pseudonymous and anonymous. For identifiable and anonymous modes, the user can simply click-and-use; for pseudonymous mode, `Mask` allows the user to maintain multiple profiles on the same device.

4.4.3 APK Rewriter

So far, we described how to build a user-level sandbox in Android that provides unlinkable runtime environments. Next, we describe how to merge our sandbox component seamlessly into an app, using the APK rewriter we developed.

Specifically, we use apktool to decompile an Android application package file (APK) into human-readable smali codes, include our sandbox component and then recompile the files back into an executable APK. This is difficult because each APK is an integral structure, and including our sandbox component is not as easy as copying all the files — we have to make sure each component serves its designed functionality at the right place, and this can be challenging especially when we need to consider the integration of UI. This brings the following technical challenges:

4.4.3.1 Sandbox Initialization

The APK rewriter needs to make sure that the sandbox component will be initialized before any component of the original application executes. This is achieved by exploiting an application base class which is provided by Android to maintain global application states. The nice property of this base class is that it is the first user-controlled component that gets initialized for any process of the app. Our APK rewriter will go through the app's existing codes and checks whether the application base class already exists. If exists, we modify the existing application base and make it a subclass of the application base defined by us; otherwise, we directly insert our application base. The sandbox initialization logic is programmed into the static code section of this application base class and is guaranteed to be the first to execute.

4.4.3.2 UI Integration

All UI elements integrated in an app are referenced with a universal unique id, indexed under *res/values/ids.xml* and *res/values/public.xml*. To integrate new UI elements into an existing app, our APK rewriter automatically tracks and assigns the empty slots within the existing ids. Moreover, the APK rewriter needs to ensure control will be returned to the app after our UI cuts in line. It works by going

Table 4.3: UI & Sandbox Management Overhead

Category	*Response Time*
Load UI	169.2 ms
Create Sandbox	68.7 ms
Run Sandbox	382.2 ms
Destroy Sandbox	26.8 ms

through the manifest file, identifying the app launcher activity and statically writing an initialization logic for the launcher activity into the `onDestroy` functions in our UI activity.

Finally, the APK rewriter also parses the manifest file to get the list of processes this app hosts. This information will be hard-coded into the smali codes of the sandbox manager to enable `Mask`'s support for multiple processes.

4.4.4 Performance Overhead

`Mask` incurs two types of performance overhead: on sandbox management when a sandbox is created, destroyed or executed; and during application runtime, after an app starts execution in a sandbox of the user's choice.

The overheads on sandbox management are measured by instrumenting a testing app with `Mask` and timers. Then we perform selected sandbox management tasks and log the output of these timers. As the results in Table 4.3 show, the most time-consuming actions in sandbox management are loading UI and running a sandbox, each taking a few hundred milliseconds. However, since these actions happen only once during a *session*, the cumulative overhead on sandbox management is still minor—far less than one second.

The overhead on apps' runtime originates from `Mask`'s user-level sandbox component when it intercepts inter-process communications (IPCs) and file system operations. To measure the overhead incurred by redirecting file system operations, we choose a benchmark app, AndroBench [6], which is designed for measuring storage

Table 4.4: Mobile App Runtime Overhead

Category	*Bench*	*Unit*	*Overhead (%)*
File	Seq Read	MB/s	$< 1\%$
	Seq Write	MB/s	1.3%
	Rand Read	MB/s	$< 1\%$
	Rand Write	MB/s	$< 1\%$
	Create	TPS	2.1%
	Delete	TPS	4.7%
Database	Insert	TPS	2.3%
	Update	TPS	9.0%
	Delete	TPS	3.4%
IPC	Filter	TPS	$< 1\%$
	Reformat	TPS	37.8%

`TPS: Transactions Per Second`

performance on Android devices. Besides the test benches included in AndroBench, we added two more tests to measure the performance of creating and deleting files. Each test bench is executed 10 times with `Mask` enabled and disabled. To evaluate the overhead caused by intercepting IPCs, we use a synthesized benchmark which contains two test benches, measuring the overheads incurred by IPCs filtering and reformatting, respectively. IPC filtering differentiates the IPCs we are interested in, such as getting device ID, from those we are not, such as getting location updates, while IPC reformatting reconstructs a low-level binary sequence into high-level objects and modifies the corresponding persistent identifiers. The IPC filtering incurs overhead to any IPC between an app and other parties, while the IPC reformatting overhead is incurred only for those IPCs that return personal or device identifying information to the app.

The results on application runtime overheads are summarized in Table 4.4. The performance degradation on file system operations is minor because the only overhead incurred is for transforming the paths in the app's original storage to the paths in the sandbox's shadow storage. This transformation is much lighter-weighted compared to file system IOs. We also found the IPC filtering overhead to be negligible, meaning that `Mask` does not affect the performance of most ‘uninteresting’ IPC calls. By contrast, the IPC reformatting overhead is significant (more than 37%) because parsing byte array into high-level objects, for example java objects, is expensive. However, since we only reformat a very small portion of all the IPCs—only those return persistent identifiers—the overall performance degradation is found negligible.

4.4.5 Applicability

`Mask` applies different *private execution modes* for mobile applications with different usage patterns (exclusive, durative or stateless). Here, we case studied the top 200 free apps in Google Play and study how many of them can be used with `Mask`. We assume a privacy-aware user who always selects the usage pattern that delivers the highest privacy level without compromising the major functionality of the app. We classify the apps according to the usage patterns and further break down the numbers into each functional category, as shown in Fig. 4.6. To sum up, 29% of the apps can be used statelessly, 43% can be used duratively and 25% have to be used exclusively. To note, `Mask` is only applicable to 97% of all the apps, excluding apps designed for system usage, such as file explorer, anti-virus software, etc. Sandboxing these apps fundamentally violates their functionalities and results in unpredictable results or even crashes.

We also conducted a study on 27 mobile users who had neither prior knowledge of the `Mask` project nor domain-specific expertise. The users are recruited by posting surveys in our university library. Our findings are summarized as follows:

- 80% of the mobile users are aware that app developers or advertising libraries may conduct user profiling. 40% of the mobile users consider this as a moderate or serious privacy threat, 45% as a minor privacy threat while 15% of them are not concerned with it at all.

- 60% of the mobile users believe maintaining multiple isolated profiles for the

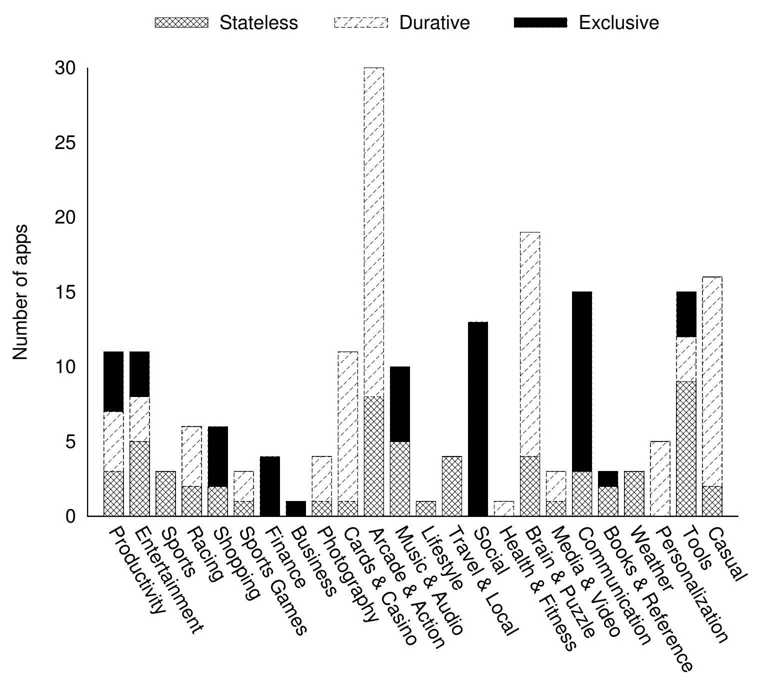


Figure 4.6: Breakdown of apps according to their usage patterns on a per-category basis

same mobile app will provide better privacy protection. Of these people, more than 70% are willing to take actions if theyre given such an option.

4.5 Related Work

There have been numerous studies on information access control on smartphones [18, 31, 33, 49, 81]. They focus on detecting and defending illegitimate collection of sensitive user information, by malicious mobile apps or third parties. Orthogonal to these studies, `Mask` focuses on *unregulated aggregation* of sensitive user information, irrespective of how it is collected. There also exist other efforts on the issue of information aggregation in the mobile ecosystem [40, 45, 86, 95, 98]. Most of them only target a restricted scenario — advertising agencies — and assume mobile apps are trustworthy. Different from these works, `Mask` considers a more general and realistic scenario and and breaks unregulated aggregation by both mobile apps, advertising agencies and network sniffers. Linkdroid [40] tries to solve a similar problem but requires extensive modifications on the smartphone OS. `Mask`, instead, utilize the usage patterns of apps and strikes a useful trade-off by delivering private execution modes in the user-level.

MoRePriv [29], πBox [60] also focused on resolving the conflict between privacy and personalization of mobile apps. MoRePriv argued that personalization should be provided by the OS instead of apps. Similarly in principle, πBox shifts the responsibility of protecting privacy from the app and its users to the platform itself. By contrast, `Mask` neither advocates a new ecosystem, nor requires modification to the existing one. It provides a practically deployable design which allows end-users to 'negotiate' with the mobile app at the client side.

4.6 Conclusion

In the current mobile app ecosystem, app usages of a user are linkable by default. This allows an interested/curious party to conduct unsolicited user profiling, targeted advertising or public surveillance. In this chapter, we designed and implemented a practical solution called `Mask`, allowing users to unlink app usages that are functionally independent of each other. Specifically, we introduce a set of *private execution modes* and give users options to specify whether and within which scope his current app session should be linkable. We presented the technical details and challenges of our user-level implementation, and evaluated the performance and applicability of `Mask`.

CHAPTER V

Continuous Authentication for Voice Assistants

5.1 Introduction

Siri, Cortana, Google Now, and Alexa are becoming our everyday fixtures. Through voice interactions, these and other voice assistants allow us to place phone calls, send messages, check emails, schedule appointments, navigate to destinations, control smart appliances, and perform banking services. In numerous scenarios such as cooking, exercising or driving, voice interaction is preferable to traditional touch interfaces that are inconvenient or even dangerous to use. Furthermore, a voice interface is even essential for the increasingly prevalent Internet of Things (IoT) devices that lack touch capabilities [76].

With sound being an open channel, voice as an input mechanism is inherently insecure as it is prone to replay, sensitive to noise, and easy to impersonate. Existing voice authentication mechanisms, such as Google's "Trusted Voice" and Nuance's "FreeSpeech" used by banks,[1] fail to provide the security features for voice assistant systems. An adversary can bypass these voice-as-biometric authentication mechanisms by impersonating the user's voice or simply launching a replay attack. Recent studies have demonstrated that it is possible to inject voice commands remotely with

[1]https://wealth.barclays.com/en$_g$b/home/international-banking/insight-research/manage-your-money/banking-on-the-power-of-speech.html

mangled voice [19, 104], wireless signals [55], or through public radio stations [72] without raising the user's attention. Even Google warns against its voice authentication feature as being insecure,[2] and some security companies [14] recommend relinquishing voice interfaces all together until security issues are resolved. The implications of attacking voice-assistant systems can be severe, ranging from information theft and financial loss [71] all the way to inflicting physical harm via unauthorized access to smart appliances and vehicles.

In this chapter, we propose `VAuth`, a novel system that provides *usable* and *continuous* authentication for voice assistant systems. As a wearable security token, it supports on-going authentication by matching the user's voice with an additional channel that provides physical assurance. `VAuth` collects the body-surface vibrations of a user via an accelerometer and continuously matches them to the voice commands received by the voice assistant. This way, `VAuth` guarantees that the voice assistant executes *only* the commands that originate from the voice of the owner. `VAuth` offers the following salient features.

Continuous Authentication `VAuth` specifically addresses the problem of continuous authentication of a speaker to a voice-enabled device. Most authentication mechanisms, including all smartphone-specific ones such as passwords, PINs, patterns, and fingerprints, provide security by proving the user's identity before establishing a session. They hinge on one underlying assumption: the user retains exclusive control of the device right after the authentication. While such an assumption is natural for touch interfaces, it is unrealistic for the case of voice assistants. Voice allows access for any third party during a communication session, rendering pre-session authentication insufficient. `VAuth` provides ongoing speaker authentication during an entire session

[2]When a user tries to enable Trusted Voice on Nexus devices, Google explicitly warns that it is less secure than password and can be exploited by the attacker with a very similar voice.

by ensuring that every speech sample recorded by the voice assistant originates from the speaker's throat. Thus, `VAuth` complements existing mechanisms of initial session authentication and speaker recognition.

Improved Security Features Existing biometric-based authentication approaches tries to reduce time-domain signals to a set of vocal features. Regardless of how descriptive the features are of the speech signal, they still represent a projection of the signal to a reduced-dimension space. Therefore, collisions are bound to happen; two different signals can result in the same feature vector. For example, Tavish *et al.* [104] fabricated mangled voice segments, incomprehensible to a human, but map to the same feature vector as a voice command so that they are recognizable by a voice assistant. Such attacks weaken the security guarantees provided by almost all voice-biometric approaches [85].

In contrast, `VAuth` utilizes an instantaneous matching algorithm to compare the entire signal from accelerometer with that of microphone in the time domain. `VAuth` splits both accelerometer and microphone signals into speech segments and proceeds to match both signals one segment at a time. It filters the non-matching segments from the microphone signal and only passes the matching ones to the voice assistant. Our theoretical analysis of `VAuth`'s matching algorithm (section 5.8) demonstrates that it prevents an attacker from injecting any command even when the user is speaking. Moreover, `VAuth` overcomes the security problems of leaked or stolen voice biometric information, such as voiceprints. A voice biometric is a lifetime property of an individual, and leaking it renders voice authentication insecure. On the other hand, when losing `VAuth` for any reason, the user has to just unpair the token and pair a new one.

Usability A user can use `VAuth` out-of-the-box as it does not require any user-specific training, a drastic departure from existing voice biometric mechanisms. It only depends on the instantaneous consistency between the accelerometer and microphone signals; therefore, it is immune to voice changes over time and in different situations, such as sickness or tiredness. `VAuth` provides its security features as long as it touches the user's skin at any position on the facial, throat, and sternum[3] areas. This allows us to incorporate `VAuth` into wearables that people are already using on a daily basis, such as eyeglasses, Bluetooth earbuds and necklaces/lockets. Our usability survey of 952 individuals revealed that users are willing to accept the different configurations of `VAuth`, especially when they are concerned about the security threats and when `VAuth` comes in the forms of which they are already comfortable.

We have implemented a prototype of `VAuth` using a commodity accelerometer and an off-the-shelf Bluetooth transmitter. Our implementation is built into the Google Now system in Android, and could easily extend to other platforms such as Cortana, Siri, or even phone banking services. To demonstrate the effectiveness of `VAuth`, we recruited 18 participants and asked each of them to issue 30 different voice commands using `VAuth`. We repeated the experiments for three wearable scenarios: eyeglasses, earbuds and necklace. We found that `VAuth`:

- delivers almost perfect results with more than 97% detection accuracy and close to 0 false positives. This indicates most of the commands are correctly authenticated from the first trial and `VAuth` only matches the command that originates from the owner;
- works out-of-the-box regardless of variation in accents, mobility patterns (still vs. jogging), or even across languages (Arabic, Chinese, English, Korean, Persian);
- effectively thwarts mangling voice attacks and successfully blocks unauthenticated voice commands replayed by an attacker or impersonated by other users;

[3]The sternum is the bone that connects the rib cage; it vibrates as a result of the speech.

and

- incurs negligible latency (an average of 300ms) and energy overhead (requiring re-charging only once a week).

The rest of the chapter is organized as follows. Section 5.2 discusses the related work while Section 5.3 provides the necessary background of human speech models. Section 5.4 states the system and threat models and Section 5.5 details the design and implementation of `VAuth`. We discuss our matching algorithm in Section 5.6, and conduct phonetic-level analysis on the matching algorithm in Section 5.7. We further study the security properties of the matching algorithm in Section 5.8 using a theoretical model. Section 5.9 evaluates `VAuth`'s effectiveness. Section 5.10 discusses `VAuth`'s features. Finally, the chapter concludes with Section 5.11.

5.2 Related Work

Smartphone Voice Assistants Many researchers have studied the security issues of smartphone voice assistants [30, 55, 88, 104]. They have also demonstrated the possibility of injecting commands into voice assistants with electromagnetic signals [55] or with a mangled voice that is incomprehensible to humans [104]. These practical attack scenarios motivate us to build an authentication scheme for voice assistants. Petracca *et al.* [88] proposed a generic protection scheme for audio channels by tracking suspicious information flows. This solution prompts the user and requires manual review for *each* potential voice command. It thus suffers from the habituation and satisficing drawbacks since it interrupts the users from their primary tasks [38].

Voice Authentication Most voice authentication schemes involve training on the user's voice samples and building a voice biometric [12, 22, 28, 57]. The biometric may depend on the user's vocal features or cultural backgrounds and requires rigorous training to perform well. There is no theoretical guarantee that they provide good security in general. Approaches in this category project the signal to a reduced-dimension space and collisions are thus inherent. In fact, most companies adopt these mechanisms for the usability benefits and claim they are not as secure as passwords or patterns [78]. Moreover, for the particular case of voice assistants, they all are subject to simple replay attacks.

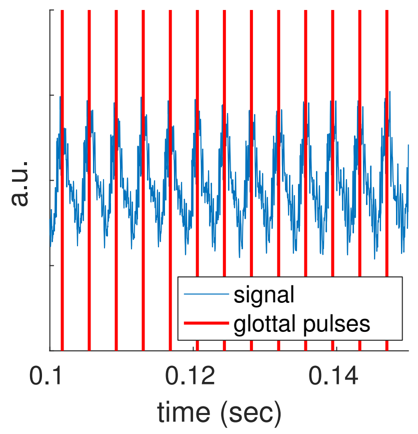
Mobile Sensing Many researchers have studied the potential applications of accelerometers for human behavior analysis [9, 70, 84, 111]. Studies show that it is possible to infer keyboard strokes [9], smartphone touch inputs [111] or passwords [9, 84] from acceleration information. There are also applications utilizing the correlation between sound and vibrations [62, 75] for health monitoring purposes. Doctors can thus detect voice disorder without actually collecting the user's daily conversations. These studies are very different from ours which focuses on *continuous* voice assistant security.

5.3 Background

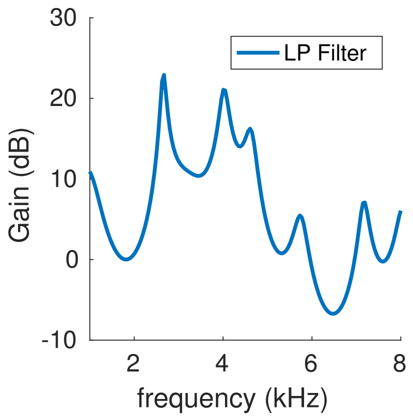
We introduce some basic concepts and terminology regarding the generation and processing of human speech, which will be referenced consistently throughout the chapter.

5.3.1 Human Speech Model

The production of human speech is commonly modeled as the combined effect of two separate processes [46]: a voice source (vibration of vocal folds) that generates the original signal and a filter (determined by the resonant properties of vocal tract including the influence of tongue and lips) that further modulates the signal. The output is a shaped spectrum with certain energy peaks, which together maps to a



(a) Source



(b) Filter

Figure 5.1: The source–filter model of human speech production using the vowel {`i:`} as an example.

specific phoneme (see Fig. 5.1b for the vowel {`i:`} – the vowel in the word "see"). This process is widely used and referred to as the *source-filter* model.

Fig. 5.1a shows an example of a female speaker pronouncing the vowel {`i:`}. The time separating each pair of peaks is the length of each glottal pulse (cycle). It also refers to the *instantaneous fundamental frequency* (f_0) variation while the user is speaking, which is the pitch of speaker's voice. The value of f_0 varies between 80 to 333Hz for a human speaker. The glottal cycle length (being the inverse of the fundamental frequency) varies 0.003sec and 0.0125sec. As the human speaker pronounces different phonemes in a particular word, the pitch changes accordingly, which becomes an important feature of speaker recognition. We utilize the fundamental frequency (f_0) as a reference to filter signals that fall outside of the human speech range.

5.3.2 Speech Recognition and MFCC

The *de facto* standard and probably the most widely used feature for speech recognition is Mel-frequency cepstral coefficients (MFCC) [63], which models the way humans perceive sounds. In particular, these features are computed on short-term windows when the signal is assumed to be stationary. To compute the MFCCs, the speech recognition system computes the short-term Fourier transform of the signal, then scales the frequency axis to the non-linear Mel scale (a set of Mel bands). Then, the Discrete Cosine Transform (DCT) is computed on the log of the power spectrum of each Mel band. This technique works well in speech recognition because it tracks the invariant feature of human speech across different users. However, it also opens the door to potential attacks: by generating mangled voice segments with the same MFCC feature, an attacker can trick the voice assistant into executing specific voice commands without drawing any attention from the user.

5.4 System and Threat Models

5.4.1 System Model

`VAuth` consists of two components. The first is an accelerometer mounted on a wearable device which can be placed on the user's chest, around the neck or on the facial area. The second component is an extended voice assistant that issues voice commands after correlating and verifying both the accelerometer signal from the wearable device and the microphone signal collected by the assistant. This system is not only compatible with smartphone voice assistants such as Siri and Google Now, but also applies to voice systems in other domains such as Amazon Alexa and phone-based authentication system used by banks. We assume the communications between the two components are encrypted. Attacks to this communication channel are orthogonal to this work. We also assume the wearable device serves as a secure token that the user will not share with others. The latter assumption is known as *security by possession*, which is widely adopted in the security field in the form of authentication rings [105], wristbands [68], or RSA SecurID. Thus, the problem of authenticating the wearable token to the user is orthogonal to `VAuth` and has been addressed elsewhere [23]. Instead, we focus on the problem of authenticating voice commands, assuming the existence of a trusted wearable device.

5.4.2 Threat Model

We consider an attacker who is interested in stealing private information or conducting unauthorized operations by exploiting the voice assistant of the target user. Typically, the attacker tries to hijack the voice assistant of the target user and deceive it into executing mal-intended voice commands, such as sending text messages to premium phone numbers or conducting bank transactions. The adversary mounts the attack by interfering with the audio channel. This does not assume the attacker has to be physically at the same location as the target. It can utilize equipment that can generate a sound on its behalf, such as radio channels or high-gain speakers. Specifically, we consider the following three categories of attack scenarios.

Scenario A – Stealthy Attack The attacker attempts to inject either inaudible or incomprehensible voice commands through wireless signals [55] or mangled voice commands [19, 104]. This attack is stealthy in the sense that the victim may not even be aware of the on-going threat. It is also preferable to the attacker when the victim has physical control or within close proximity of the voice assistant.

Scenario B – Biometric-override Attack The attacker attempts to inject voice commands [85] by replaying a previously recorded clip of the victim's voice, or by impersonating the user's voice. This attack can have a very low technical barrier: we found that by simply mimicking the victim's voice, an attacker can bypass the Trusted Voice feature of Google Now within five trials, even when the attacker and the victim are of different genders.

Scenario C – Acoustic Injection Attack The attacker can be more advanced, trying to generate a voice that has a direct effect on the accelerometer [102]. The intention is to override `VAuth`'s verification channel with high energy vibrations. For example, the attacker can play very loud music which contains embedded patterns of voice commands.

5.5 VAuth

We now present the high-level design of `VAuth`, describe our prototype implementation with Google Now, and elaborate on its usability aspects.

5.5.1 High-Level Overview

`VAuth` has two components: (1) a wearable component, responsible for collecting and uploading the accelerometer data, and (2) a voice assistant extension, responsible for authenticating and launching the voice commands. The first component easily incorporates into existing wearable products, such as earbuds/earphones/headsets, eyeglasses, or necklaces/lockets. The usability aspect of `VAuth` will be discussed later in this sectiontoken that the user does not share with others. When a user triggers the voice assistant, for example by saying "OK, Google" or "Hey, Siri", our voice assistant extension will fetch accelerometer data from the wearable component, correlate it with signals collected from microphone and issue the command only when there is a match. Fig. 5.2 depicts the information flows in our system. To reduce the processing burden on the user's device, the matching does not take place on the device (that runs the voice assistant), but rather at the server side. The communication between the wearable component and the voice assistant takes place over Bluetooth BR/EDR [15]. Bluetooth Classic is an attractive choice as a communication channel, since it has a relatively high data rate (up to 2Mbps), is energy-efficient, and enables secure communication through its pairing procedure.

The design of `VAuth` is modular and compatible with most voice assistant systems. One can thus customize any component in Fig. 5.2 to optimize functionality,

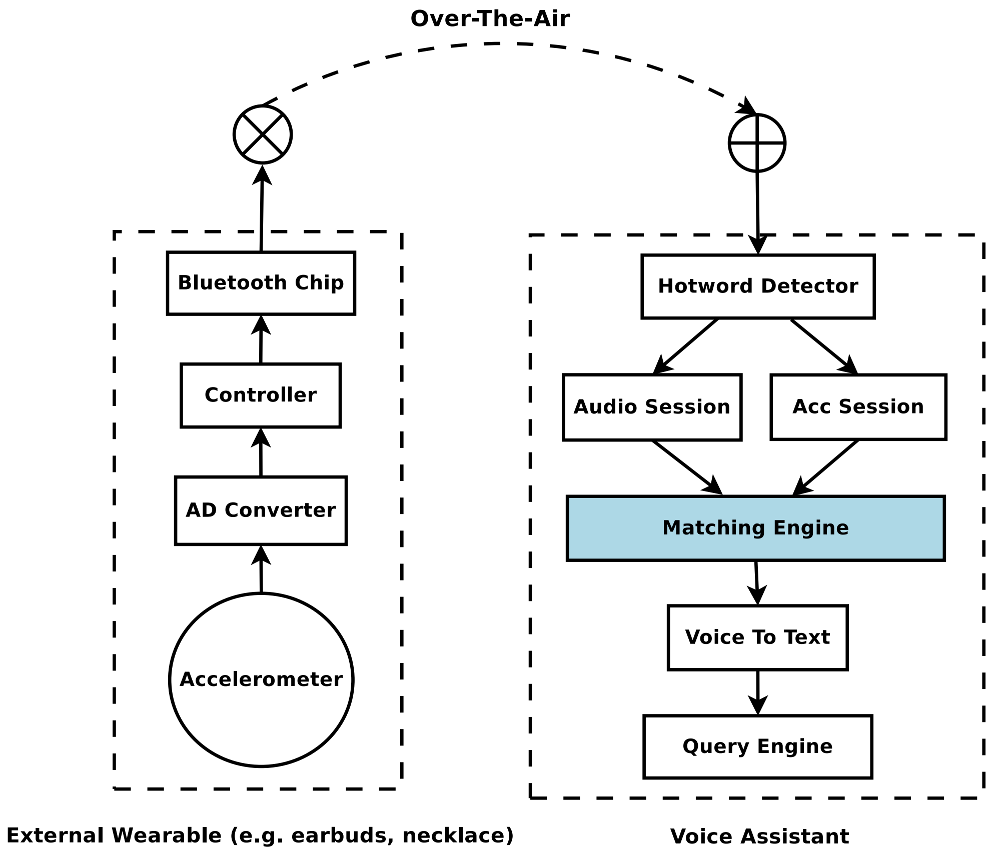
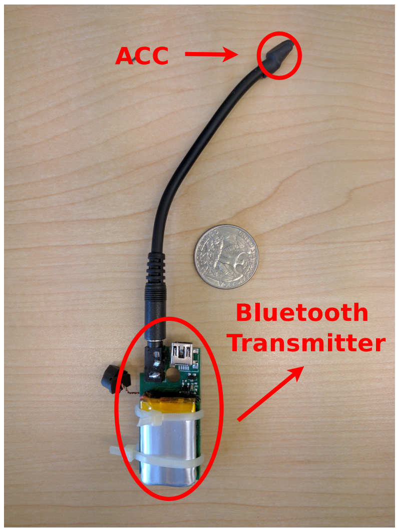
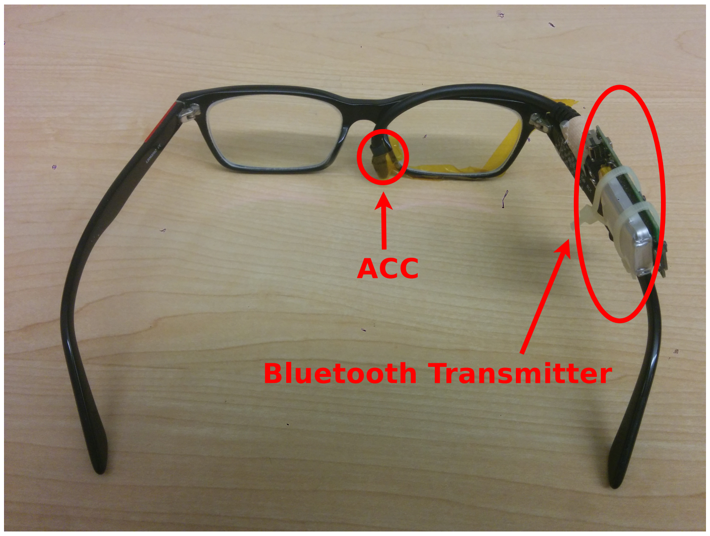


Figure 5.2: The high-level design of `VAuth`, consisting of the wearable and the voice assistant extension.



(a) Wireless



(b) Eyeglasses

Figure 5.3: Our prototype of `VAuth`, featuring the accelerometer chip and Bluetooth transmitter, (a) compared to US quarter coin and (b) attached to a pair of eyeglasses belonging to one of the authors.

performance or usability. Here, we elaborate how to integrate this into an existing voice assistant, using Google Now as an example.

5.5.2 Prototype

We first elaborate on our design of the wearable component. We use a Knowles BU-27135 miniature accelerometer with the dimension of only 7.92×5.59×2.28mm so that it can easily fit in any wearable design. The accelerometer uses only the z-axis and has an analog bandwidth of 11kHz, enough to capture the bandwidth of a speech signal. We utilize an external Bluetooth transmitter that provides Analog-to-Digital Conversion (ADC) and Bluetooth transmission capabilities to the voice assistant extension. To reduce energy consumption, *BinderCracker*starts streaming the accelerometer signal only upon request from the voice assistant. Our prototype communicates the microphone and accelerometer signals to a Matlab-based server which performs the matching and returns the result to the voice assistant. Fig. 5.3 depicts our wireless prototype standalone, and attached to a pair of eyeglasses.

Our system is integrated with Google Now voice assistant to enable voice command authentication. `VAuth` starts execution immediately after the start of a voice session (right after "OK Google" is recognized). It blocks the voice assistant's command execution after the voice session ends until the matching result becomes available. If the matching fails, `VAuth` kills the voice session. To achieve its functionality, `VAuth` intercepts both the `HotwordDetector` and the `QueryEngine` to establish the required control flow.

Our voice assistant extension is implemented as a standalone user-level service. It is responsible for retrieving accelerometer signals from the wearable device, and sending both accelerometer and microphone to our Matlab-based server for analysis. The user-level service provides two RPC methods, `start` and `end`, which are triggered by the events generated when the hotword "OK Google" is detected, and when the query (command) gets executed, respectively. The first event can be observed by filtering the Android system logs, and we intercept the second by overriding the Android IPC mechanisms, by filtering the Intents sent by Google Now. Also, since some Android devices (e.g., Nexus 5) do not allow two apps to access the microphone at the same time, we need to stream the voice signal retrieved by the voice assistant to our user-level service. We solve this by intercepting the `read` method in the `AudioRecord` class. Whenever Google Now gets the updated voice data through this interface, it will forward a copy of the data to our user-level service via another RPC method.

Note that the modifications and interceptions above are necessary only because we have no access to the Google Now source. The incorporation of `VAuth` is straightforward in the cases when developers try to build/extend their voice assistant.

5.5.3 Usability

`VAuth` requires the user to wear a security-assisting device. There are two gen-

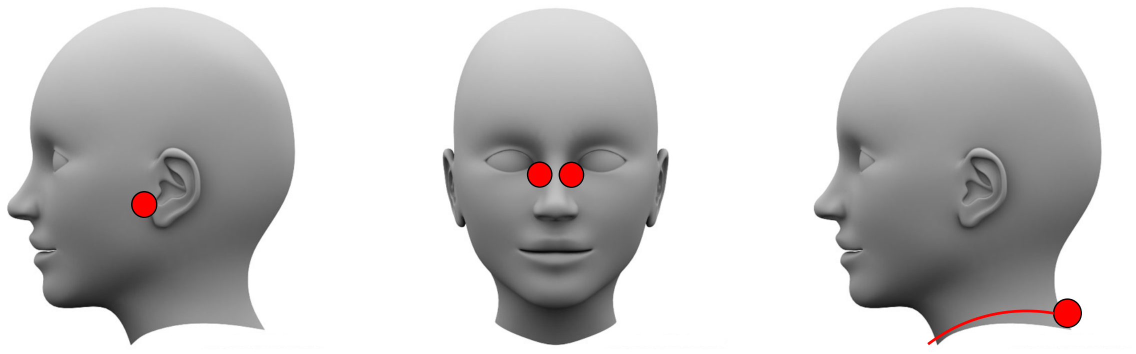

(a) Earbuds (b) Eyeglasses (c) Necklace

Figure 5.4: The wearable scenarios supported by `VAuth`.

eral ways to meet this requirement. The first is to ask users to wear an additional device for security, while the other is to embed `VAuth` in existing wearable products that the users are already comfortable with in their daily lives. We opted for the latter as security has always been a secondary concern for users [107]. Our prototype supports three widely-adopted wearable scenarios: earbuds/earphones/headsets, eyeglasses, and necklace/lockets. Fig. 5.4 shows the positions of the accelerometer in each scenario. We select these areas because they have consistent contact with the user's body. While `VAuth` performs well on all facial areas, shoulders and the sternal surface, we only focus on the three positions shown in Fig. 5.4 since they conform with widely-adopted wearables.

We have conducted a usability survey to study the users' acceptance of the different configurations of `VAuth`. We surveyed 952 individuals using Amazon Mechanical Turk. We restricted the respondent pool to those from the US with previous experience with voice assistants. We compensated each respondent with $0.5 for their participation. Of the respondents, 40% are female, 60% are employed full-time, and 67% have an education level of associate degree or above. Our respondents primarily use voice assistants for information search (70%), navigation (54%), and communication (47%). More than half (58%) of them reported using a voice assistant at least once a week.

Survey Design: We follow the USE questionnaire methodology [64] to measure the usability aspects of `VAuth`. We use a 7-point Likert scale (ranging from Strongly Disagree to Strongly Agree) to assess the user's satisfaction with a certain aspect or configuration of `VAuth`. We pose the questions in the form of how much the respondent agrees with a certain statement, such as: *I am willing to wear a necklace that contains the voice assistant securing technology.* Below, we report a favorable result as the portion of respondents who answered a question with a score higher than 4 (5,6,7) on the 7-point scale. Next to each result, we report the portion of those surveyed, between brackets, who answered the question with a score higher than 5 (6 or 7).

The survey consists of three main parts that include: demographics and experience with voice assistants, awareness of the security issues, and the perception towards `VAuth`. In Section 5.9, we will report more on the other usability aspects of `VAuth`, such as matching accuracy, energy, and latency.

Security Awareness: We first asked the respondents about their opinion regarding the security of voice assistants. Initially, 86% (63%) of the respondents indicate that they think the voice assistants are secure. We then primed the respondents about the security risks associated with voice assistants by iterating the attacks presented in Section 5.4. Our purpose was to study the perception of using `VAuth` from individuals who are already aware of the security problems of voice assistants.

After the priming, the respondents' perceptions shifted considerably. 71% (51%) of the respondents indicate that attacks to voice assistants are dangerous, and 75%(52%) specified that they would take steps to mitigate the threats. Almost all of the latter belong to the set of respondents who now regard these attacks as dangerous to them.

Wearability: In the last part of the survey, we ask the participants about their preferences for wearing `VAuth` in any of the three configurations of Fig. 5.4. We have the following takeaways from the analysis of survey responses.

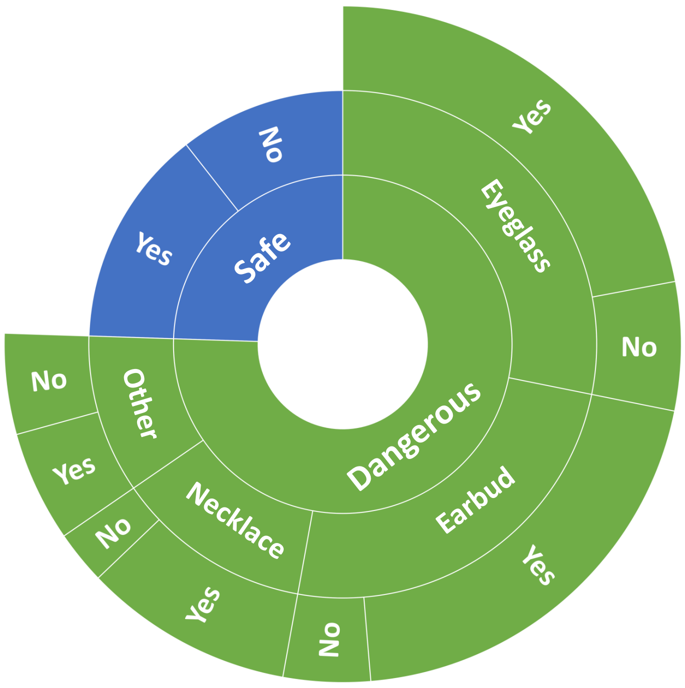


Figure 5.5: A breakdown of respondents' wearability preference by security concern and daily wearables. *Dangerous* and *Safe* refer to participants' attitudes towards the attacks to voice assistants after they've been informed; the *Dangerous* category is further split according to the wearables that people are already wearing on a daily basis; *Yes* and *No* refer to whether participants are willing to use `VAuth` in at least one of three settings we provided.

- 70%(47%) of the participants are willing to wear at least one of `VAuth`'s configurations to provide security protection. These respondents are the majority of those who are strongly concerned about the security threats.

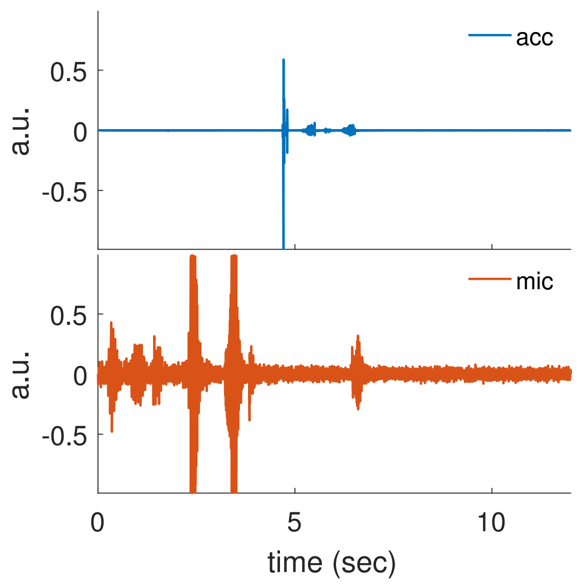
- 48% (29%) of the respondents favored the earbuds/earphone/headset option, 38% (23%) favored the eyeglasses option and 35% (19%) favored the necklace/locket option. As expected, the findings fit the respondents' wearables in their daily lives. 71% of the respondents who wear earbuds on a daily basis favored that option for `VAuth`, 60% for eyeglasses and 63% for the necklace option.

- There is no discrepancy in the wearable options among both genders. The gender distribution of each wearable option followed the same gender distribution of the whole respondent set.

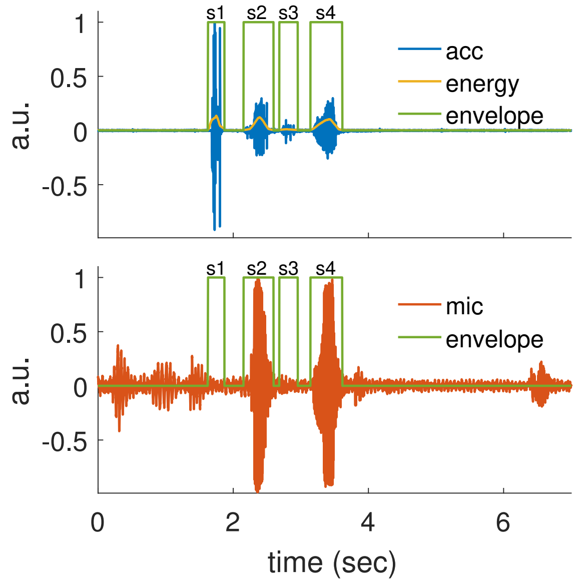
- More than 75% of the users are willing to pay $10 more for a wearable equipped with this technology while more than half are willing to pay $25 more.

- Respondents were concerned about the battery life of `VAuth`. A majority of 73% (81%) can accommodate charging once a week, 60% (75%) can accommodate once per 5 days, and 38% (58%) can accommodate once each three days. In Section 5.9, we show that the energy consumption of `VAuth` matches the respondents' requirements.

Fig. 5.5 presents a breakdown of the major findings in our usability survey. These results demonstrate that users are willing to accept the different configurations of `VAuth`, especially when they are concerned about the privacy/security threats and when `VAuth` comes in the forms of which they are already comfortable with.



(a) Raw input signals



(b) Energy envelopes

Figure 5.6: Pre-processing stage of VAuth's matching.

5.6 Matching Algorithm

The matching algorithm of VAuth (highlighted in Fig. 5.2) takes as input the speech and vibration signals along with their corresponding sampling frequencies. It outputs a decision value indicating whether there is a match between the two signals as well as a "cleaned" speech signal in case of a match. VAuth performs the matching in three stages: *pre-processing*, *speech segments analysis*, and *matching decision*.

In what follows, we elaborate on VAuth's matching algorithm using a running example of a male speaker recording the two words: "cup" and "luck" with a short pause between them. The speech signal is sampled by an accelerometer from the lowest point on the sternum at 64kHz and recorded from a built-in laptop microphone at a sampling frequency of 44.1kHz, 50cm away from the speaker.

5.6.1 Pre-processing

First, `VAuth` applies a highpass filter, with cutoff frequency at 100 Hz, to the accelerometer signal. The filter removes all the artifacts of the low-frequency user movement to the accelerometer signal (such as walking or breathing). We use 100Hz as a cutoff threshold because humans cannot generate more than 100 mechanical movements per second. `VAuth` then re-samples both accelerometer and microphone signals to the same sampling rate while applying a low-pass filter at 4kHz to prevent aliasing. We choose a sampling rate of 8kHz that preserves most acoustic features of the speech signal and reduces the processing load. Thus, `VAuth` requires an accelerometer of bandwidth larger than 4kHz. Then `VAuth` applies Fig. 5.6a shows both raw signals immediately after both signals are filtered and resampled. As evident from the figure, the accelerometer signal has a high-energy spike due to the sudden movement of the accelerometer (e.g., rubbing against the skin), and small energy components resulting from speech vibrations. On the other hand, the speech signal has two high-energy segments along with other lower-energy segments corresponding to background noise.

Second, `VAuth` normalizes the magnitude of both signals to have a maximum magnitude of unity, which necessitates removal of the spikes in the signals. Otherwise, the lower-energy components referring to the actual speech will not be recovered. The matching algorithm computes a running average of the signal's energy and enforces a cut-off threshold, keeping only the signals with energy level within the moving average plus six standard deviation levels.

After normalizing the signal magnitude, as shown in the top plot of Fig. 5.6b, `VAuth` aligns both signals by finding the time shift that results in the maximum cross correlation of both signals. Then, it truncates both signals to make them have the same length. Note that `VAuth` does not utilize more sophisticated alignment algorithms such as Dynamic Time Warping (DTW), since they remove timing information critical to the signal's pitch and they also require a higher processing load. Fig. 5.6b shows both accelerometer and microphone signals aligned and normalized.

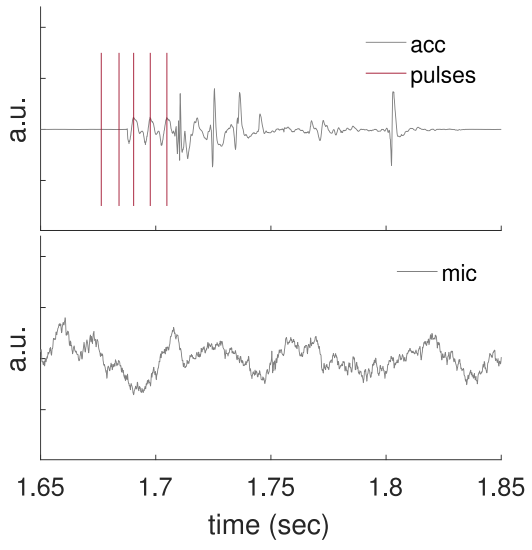
The next pre-processing step includes identification of the energy envelope of the accelerometer signal and then its application to the microphone signal. `VAuth` identifies the parts of the signal that have a significant signal-to-noise ratio (SNR). These are the "bumps" of the signal's energy as shown in the top plot of Fig. 5.6b. The energy envelope of the signal is a quantification of the signal's energy between 0 and 1. In particular, the portions of the signal with average energy exceeding 5% of maximum signal energy map to 1, and other segments map to 0. This results in four energy segments of the accelerometer signal of Fig. 5.6b. The thresholds for energy detection depend on the average noise level (due to ADC chip's sampling and quantization) when the user is silent. We chose these thresholds after studying our wireless prototype's Bluetooth transmitter.

Finally, `VAuth` applies the accelerometer envelope to the microphone signal so that it removes all parts from the microphone signal that did not result from body vibrations, as shown in the bottom plot of Fig. 5.6b. This is the first real step towards providing the security guarantees. In most cases, it avoids attacks on voice assistant systems when the user is not actively speaking. Inadvertently, it improves the accuracy of the voice recognition by removing background noise and sounds from the speech signals that could not have been generated by the user.

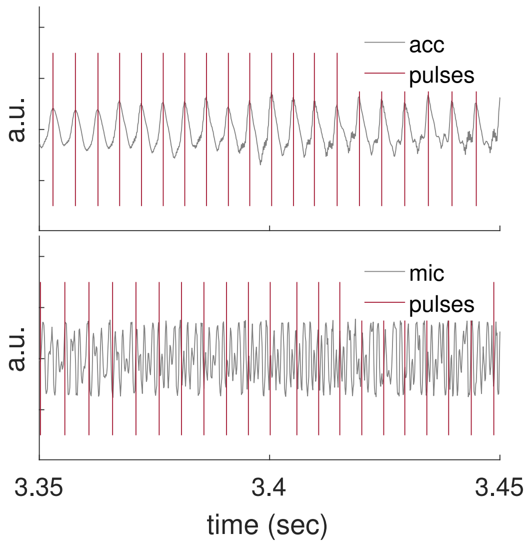
5.6.2 Per-Segment Analysis

Once it identifies high-energy segments of the accelerometer signal, `VAuth` starts a segment-by-segment matching. Fig. 5.6b shows four segments corresponding to the parts of the signal where the envelope is equal to 1.

For each segment, `VAuth` normalizes the signal magnitude to unity to remove the effect of other segments, such as the effect of the segment *s1* in Fig. 5.6b. This



(a) Non-matching segment *s1*



(b) matching segment *s4*

Figure 5.7: Per-segment analysis stage of `VAuth`.

serves to make the energy content of each segment uniform, which will elaborate on later in Section 5.8. `VAuth` then applies the approach of Boersma [16] to extract the glottal cycles from each segment. The approach relies on the identification of periodic patterns in the signal as the local maxima of the auto-correlation function of the signal. Thus, each segment is associated with a series of glottal pulses as shown in Fig. 5.7. `VAuth` uses information about the segment and the corresponding glottal pulses to filter out the segments that do not correspond to human speech and those that do not match between the accelerometer and microphone signals as follows.

1. If the length of the segment is less than 20ms, the length of a single phoneme, then `VAuth` removes the segment from both accelerometer and microphone signals. Such segments might arise from sudden noise.

2. If the segment has no identifiable glottal pulses or the length of the longest continuous sequence of glottal pulses is less than 20ms (the duration to pronounce a single phoneme), then `VAuth` also removes the segment. Fig. 5.7a shows the segment "s1" at a higher resolution. It only contains five pulses which could

not have resulted from a speech.

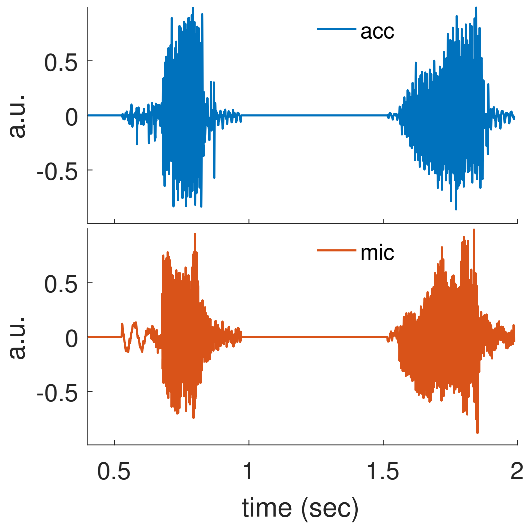
3. If the average glottal cycle of the accelerometer segment is larger than 0.003sec or smaller than 0.0125sec, then `VAuth` removes the segment from both signals. This refers to the case of the fundamental frequency falling outside the range of [80Hz, 333 Hz] which corresponds to the human speech range.

4. If the average relative distance between glottal pulse sequence between the accelerometer and microphone segments is higher than 25%, then `VAuth` removes the segment from both signals. This refers to the case of interfered speech (e.g., attacker trying to inject speech); the instantaneous pitch variations should be similar between the accelerometer and microphone [74] in the absence of external interference. For example, it is evident that the pitch information is very different between the accelerometer and microphone of Fig. 5.7a.

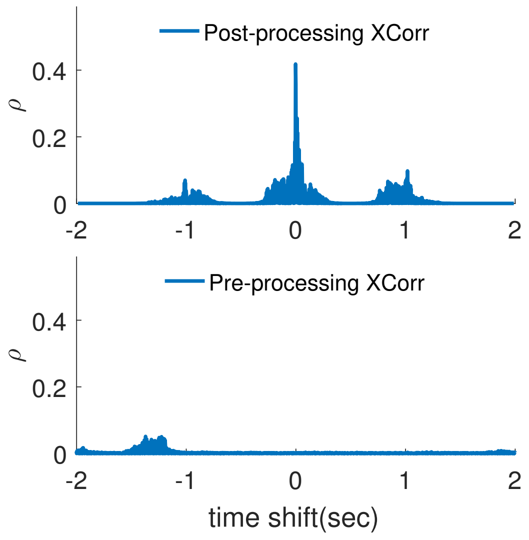
After performing all the above filtering steps, `VAuth` does a final verification step by running a normalized cross correlation between the accelerometer and microphone segments. If the maximum correlation coefficient falls inside the range [-0.25,0.25], then the segments are discarded. We use this range as a conservative way of specifying that the segments do not match (correlation coefficient close to zero). The correlation is a costlier operation but is a known metric for signal similarity that takes into consideration all the information of the time-domain signals. For example, the segment "s4" depicted in Fig. 5.7b shows matching pitch information and a maximum cross-correlation coefficient of 0.52.

5.6.3 Matching Decision

After the segment-based analysis finishes, only the "surviving" segments comprise the final accelerometer and microphone signals. In Fig. 5.8a, only the segments "s2" and "s4" correspond to matching speech components. It is evident from the bottom



(a) Output signals



(b) Cross correlation

Figure 5.8: Matching decision stage of `VAuth`'s matching.

plot that the microphone signal has two significant components referring to each word.

The final step is to produce a matching decision. `VAuth` measures the similarity between the two signals by using the normalized cross-correlation, as shown in the top plot of Fig. 5.8b. `VAuth` cannot just perform the cross-correlation on the input signals before cleaning. Before cleaning the signal, the cross-correlation results do not have any real indication of signal similarity. Consider the lower plot of Fig. 5.8b, which corresponds to the cross-correlation performed on the original input signals of Fig. 5.6a. As evident from the plot, the cross-correlation shows absolutely no similarity between the two signals, even though they describe the same speech sample.

Instead of manually constructing rules that map the cross-correlation vector to a *matching* or *non-matching* decision, we opted to utilize a machine learning-based classifier to increase the accuracy of `VAuth`'s matching. Below, we elaborate on the three components of `VAuth`'s classifier: the feature set, the machine learning algorithm and the training set.

Feature Set In general, the feature vector comprises the normalized cross-correlation values ($h(t)$) of the final accelerometer and microphone signals. However, we need to ensure that the structure of the feature vector is uniform across all matching tasks. To populate the feature vector, we identify the maximum value of $h(t)$, and then uniformly sample 500 points to the left and another 500 to the right of the maximum. We end up with a feature vector containing 1001 values, centered at the maximum value of the normalized cross-correlation.

Formally, if the length of $h(t)$ is t_e, let $t_m = \underset{t}{arg\,max}|h(t)|$. Then, the left part of the feature vector is $h_l[n] = h(\frac{n.t_m}{500}),\ 1 < n < 500$. The right part of the feature vector is $h_r[n] = h(t_m + \frac{n.(t_e - t_m)}{500}), 1 < n < 500$. The final feature vector can then be given as $h[n] = h_l[n] + h(t_m).\delta[n - 501] + h_r[n - 502]$.

Classifier We opted to use SVM as the classifier thanks to its ability to deduce linear relations between the cross-correlation values that define the feature vector. We utilize Weka [44] to train an SVM using the Sequential Minimal Optimization (SMO) algorithm [89]. The SMO algorithm uses a logistic calibrator with neither standardization nor normalization to train the SVM. The SVM utilizes a polynomial kernel with the degree equal to 1. We use the trained model in our prototype to perform the online classification.

Training Set Here, it is critical to specify that our training set has been generated offline and is user-agnostic; we performed the training only once. We recorded (more on that in Section 5.7) all 44 English phonemes (24 vowels and 20 consonants) from one of the authors at the lower sternum position using both the accelerometer and microphone. Hence, we have 44 accelerometer ($acc(i)$) and microphone ($mic(i)$) pair of recordings corresponding for each English phoneme. To generate the training set, we ran `VAuth`'s matching over all 44 $\times$ 44 accelerometer and microphone recordings to generate 1936 initial feature vectors, (fv), and their labels as follows:

$$\forall i : 1 \leq i \leq 44; \quad \forall j : 1 \leq j \leq 44;$$
$$fv[j + 44(i-1)] = match(acc(i), mic(j))$$
$$label[j + 44(i-1)] = 1_{i=j}.$$

The generated training dataset contains only 44 vectors with positive labels. This might bias the training process towards the majority class ($label = 0$). To counter this effect, we amplified the minority class by replicating the vectors with positive labels five times. The final training set contains 236 vectors with positive labels and 1892 vectors with negative labels. We use this training set to train the SVM, which, in turn, performs the online classification.

`VAuth`'s classifier is trained *offline*, only *once* and *only* using a single training set. The classifier is thus agnostic of the user, position on the body and language. In our user study and rest of the evaluation of Section 5.9, this (same) classifier is used to perform all the matching. To use `VAuth`, the user *need not* perform any initial training.

After computing the matching result, `VAuth` passes the final (cleaned and normalized) microphone signal to the voice assistant system to execute the speech recognition and other functionality.

5.7 Phonetic-Level Analysis

We evaluate the effectiveness of our matching algorithm on phonetic-level matchings/authentications. The International Phonetic Alphabet (IPA) standardizes the representation of sounds of oral languages based on the Latin alphabet. While the number of words in a language, and therefore the sentences, can be uncountable, the number of phonemes in the English language are limited to 44 vowels and consonants. By definition, any English word or sentence, as spoken by a human, is necessarily a combination of those phonemes [61]; A phoneme[4] represents the smallest unit of perceptual sound. Our phonetic-level evaluation represents a baseline of `VAuth`'s operation. Table 5.1 lists 20 vowels and 24 consonants p honemes, with two words representing examples of where the phonemes appear.

Table 5.1: The IPA chart of English phonetics.[5]

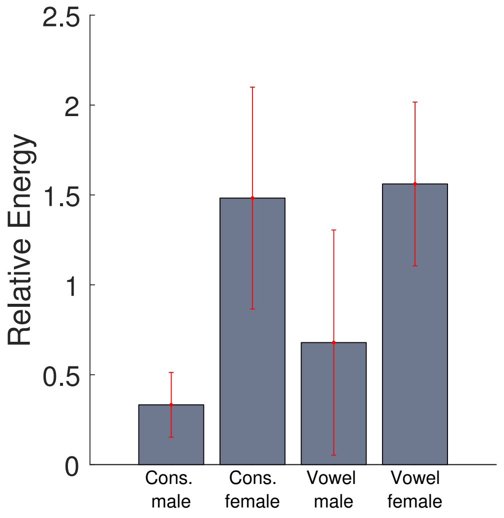
Vowel	**Examples**	**Consonants**	**Examples**
ʌ	C**U**P, L**U**CK	b	**B**AD, LA**B**
ɑː	**A**RM, F**A**THER	d	**D**I**D**, LA**D**Y
æ	C**A**T, BL**A**CK	f	**F**IND, I**F**
e	M**E**T, B**E**D	g	**G**IVE, FLA**G**
ə	AW**A**Y, CIN**E**M**A**	h	**H**OW, **H**ELLO
ɜː^r	T**UR**N, L**EAR**N	j	**Y**ES, **Y**ELLOW
ɪ	H**I**T, S**I**TT**I**NG	k	**C**AT, BA**CK**
iː	S**EE**, H**EA**T	l	**L**EG, **L**ITTLE
ɒ	H**O**T, R**O**CK	m	**M**AN, LE**M**ON
ɔː	C**A**LL, F**OU**R	n	**N**O, TE**N**
ʊ	P**U**T, C**OU**LD	ŋ	SI**NG**, FI**NG**ER
uː	BL**UE**, F**OO**D	p	**P**ET, MA**P**
ɑɪ	F**I**VE, **EYE**	r	**R**ED, T**R**Y
ɑʊ	N**OW**, **OU**T	s	**S**UN, MI**SS**
eɪ	S**A**Y, **EI**GHT	ʃ	**SH**E, CRA**SH**
oʊ	G**O**, H**O**ME	t	**T**EA, GE**TT**ING
ɔɪ	B**OY**, J**OI**N	tʃ	**CH**ECK, **CH**UR**CH**
eə^r	WH**ERE**, **AIR**	θ	**TH**INK, BO**TH**
ɪə^r	N**EAR**, H**ERE**	ð	**TH**IS, MO**TH**ER
ʊə^r	P**URE**, T**OUR**IST	v	**V**OICE, FI**V**E
-	-	w	**W**ET, **W**INDO**W**
-	-	z	**Z**OO, LA**Z**Y
-	-	ʒ	PLEA**S**URE, VI**S**ION
-	-	dʒ	**J**UST, LAR**GE**

We study if `VAuth` can correctly match the English phoneme between the accelerometer and microphone (true positives), and whether it mistakenly matches phoneme samples from accelerometer to other phoneme samples from the microphone (false positives).

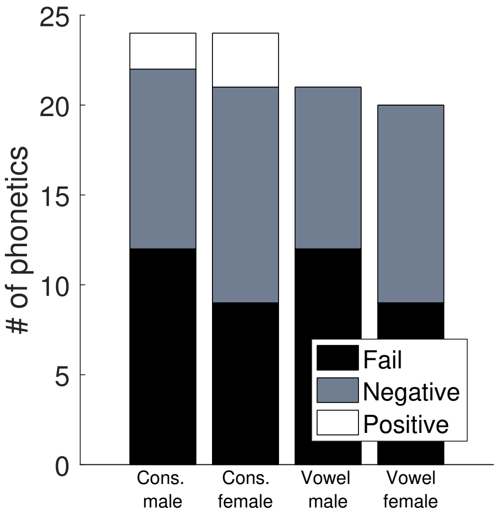
We recruited two speakers, a male and a female, to record the 44 examples listed in Table 5.1. Each example comprises two words, separated by a brief pause, both representing a particular phoneme. We asked the speaker to say both words, not just the phoneme, as it is easier for the speaker to pronounce the phoneme in the context of a word. Both participants were 50cm away from the built-in microphone of an HP

[4]https://www.google.com/search?q=define:phonemes

[5]copied from: `http://www.antimoon.com/resources/phonchart2008.pdf`



(a) Relative energy of accelerometer signal to microphone signal



(b) ASR accuracy for accelerometer signal

Figure 5.9: Analysis of the vibrations received by the accelerometer.

workstation laptop. At the same time, both speakers were wearing `VAuth`, with the accelerometer taped to the sternum. The microphone was sampled at 44100 Hz, and the accelerometer at 64000 Hz.

5.7.1 Accelerometer Energy & Recognition

Phonemes originate from a possibly different part of the chest-mouth-nasal area. In what follows, we show that each phoneme results in vibrations that the accelerometer chip of `VAuth` can register, but does not retain enough acoustic features to substitute a microphone speech signal for the purpose of voice recognition. This explains our rationale for employing the matching-based approach.

We perform the pre-processing stage of `VAuth`'s matching algorithm to clean both accelerometer and microphone signals for each phoneme. After normalizing both signals to a unity magnitude, we compute the accelerometer signal's energy relative to that of the microphone. Fig. 5.9a depicts the average relative energy of the vowel and consonants phonemes for both the female and male speakers.

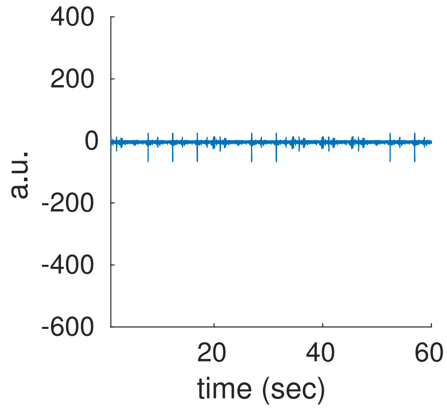
All phonemes register vibrations, with the minimum relative energy (14%) coming from the ɔɪ (the pronunciation of "oy" in "boy") phoneme of the male speaker. It is also clear from the figure that there is a low discrepancy of average relative energy between vowels and consonants for the same speaker. Nevertheless, we notice a higher discrepancy between the two speakers for the same phoneme. The female speaker has a shorter distance between the larynx and lowest point of the sternum, and she has a lower body fat ratio so that the chest skin is closer to the sternum bone. It is worth noting that the energy reported in the figure does not represent the energy at the time of the recording but after the initial processing and normalization. This explains why in some cases the accelerometer energy exceeds that of the microphone.

While the accelerometer chip senses considerable energy from the chest vibrations, it cannot substitute for the microphone. To confirm this, we passed the recorded and cleaned accelerometer samples of all phonemes for both speakers to the Nuance Automatic Speaker Recognition (ASR) API [82]. Fig. 5.9b shows the breakdown of voice recognition accuracy for the accelerometer samples by phoneme type and speaker. Clearly, a state-of-the-art ASR engine fails to identify the actual spoken words. In particular, for about half of the phonemes for both speakers, the ASR fails to return any result. Nuance API returns three suggestions for each accelerometer sample for the rest of the phonemes. These results do not match any of the spoken words. In only three cases for consonants phonemes for both speakers, the API returns a result that matches at least one of the spoken words.

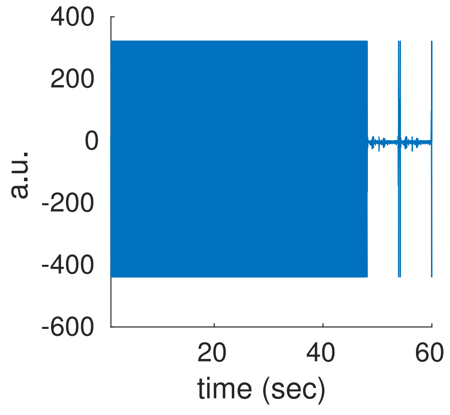
The above indicates that existing ASR engines cannot interpret the often low-fidelity accelerometer samples, but it does not indicate that ASR engines cannot be retrofitted to recognize samples with higher accuracy. This will, however, require significant changes to deploying and training these systems. On the other hand, `VAuth` is an entirely client-side solution that requires no changes to the ASR engine or the voice assistant system.

Table 5.2: The detection accuracy of `VAuth` for the English phonemes.

microphone	**accelerometer**	**TP (%)**	**FP (%)**
consonants	consonants	90	0.2
consonants	vowels	-	1.0
vowels	consonants	-	0.2
vowels	vowels	100	1.7
all	all	94	0.7



(a) Low energy and white noise



(b) High energy and periodic noise

Figure 5.10: Examples of tested noise signals.

5.7.2 Phonemes Detection Accuracy

We then evaluate the accuracy of detecting each phoneme for each speaker as well as the false positive across phonemes and speakers. In particular, we run `VAuth` to match each accelerometer sample (88 samples — corresponding to each phoneme and speaker) to all the collected microphone samples; each accelerometer sample must match only one microphone sample. Table 5.2 shows the matching results.

First, we match the consonant phonemes across the two speakers as evident in the first row. The true positive rate exceeds 90%, showing that `VAuth` can correctly match the vast majority of consonant phonemes. This is analogous to a low false negative rate of less than 10%. Moreover, we report the false positive rate which indicates the instances where `VAuth` matches an accelerometer sample to the inappropriate microphone sample. As shown in the same figure, the false positive rate is nearly zero.

Having such a very low false positive rate highlights two security guarantees that `VAuth` offers. It does not mistake a phoneme as another even for the same speaker. Recall that pitch information is widely used to perform speaker recognition, as each person has unique pitch characteristics that are independent of the speech. `VAuth` overcomes pitch characteristics similarity and is able to distinguish the different phonemes as spoken by the same speaker.

Moreover, `VAuth` successfully distinguishes the same phoneme across the two speakers. A phoneme contains speaker-independent features. `VAuth` overcomes these similar features to effectively identify each of them for each speaker. The fourth row of Table 5.2 shows comparable results when attempting to match the vowel phonemes for both speakers.

The second and third rows complete the picture of phoneme matching. They show the matching results of the vowel phonemes to the consonant phonemes for both speakers. Both rows do not contain true positive values as there are no phoneme matches. Nevertheless, one must notice the very low false positive ratio that confirms the earlier observations. Finally, the fifth row shows results of matching all the accelerometer samples to all the microphone samples. The true positive rate is 93%, meaning that `VAuth` correctly matched 82 accelerometer samples matched to their microphone counterparts. Moreover, the false positive rate was only 0.6%.

5.7.3 Idle Detection Accuracy

Last but not least, we evaluate another notion of false positives: `VAuth` mistakenly matches external speech to a silent user. We record idle (the user not actively speaking) segments from `VAuth`'s accelerometer and attempt to match them to the recorded phonemes of both participants. We considered two types of idle segments: the first contains no energy from speech or other movements (Fig. 5.10a), while the other contains significant abrupt motion of the accelerometer resulting in recordings with high energy spikes (similar to the spike of Fig. 5.6a). We also constructed a high energy noise signal with periodic patterns as shown in Fig. 5.10b.

We execute `VAuth` over the different idle segments and microphone samples and recorded the false matching decisions. In all of the experiments, we did not observe any occurrence of a false matching of an idle accelerometer signal to any phoneme from the microphone for both speakers. As recorded phonemes are representative of all possible sounds comprising the English language, we can be confident that the false positive rate of `VAuth` is zero in practice for silent users.

5.8 Theoretical Analysis

In this section, we highlight the effectiveness of the per-segment analysis of `VAuth`'s matching algorithm in preventing an attacker from injecting commands. We also show that our matching algorithm ensures the phonetic-level results constitute a lower-bound of sentence-level matching. We provide a formal analysis of this property, which will be further supported in the next section using real user studies.

5.8.1 Model

We analyze the properties of `VAuth`'s matching algorithm which takes as inputs two signals $f(t)$ and $g(t)$ originating from the accelerometer and microphone, respectively. It outputs a final matching result that is a function of normalized cross-correlation of $f(t)$ and $g(t)$: $h(t) = \frac{f(t) \star g(t)}{E}$, where $E = \sqrt{\|f(t)\|.\|g(t)\|}$, $\star$ denotes the cross-correlation operator, and $\|\cdot\|$ is the energy of the signal (autocorrelation evaluated at 0). For the simplicity of the analysis, we will focus on the most important feature of $h(t)$, its maximum value. We can then define `VAuth`'s binary matching function,

$v(f, g)$, as:

$$v(f,g) = \begin{cases} v = 0, & \text{if } 0 \leq m = \max(|h(t)|) \leq th \\ v = 1, & \text{if } th < m = \max(|h(t)|) \leq 1. \end{cases} \quad (5.1)$$

Each of the input signals comprises a set of segments, which could refer to the phonemes making up a word or words making up a sentence, depending on the granularity of the analysis. Let $f_i(t)$ and $g_i(t)$ be the i^{th} segments of $f(t)$ and $g(t)$, respectively. We assume that maximum length of a segment is τ, such that $f_i(t) = 0, t \in (-\infty, 0] \bigcup [\tau, +\infty)$. We can then rewrite $f(t)$ as $f(t) = \sum_{i=1}^{n} f_i(t - i\tau)$; the same applies for $g(t)$.

One can view the cross-correlation operation as sliding the segments $g_i(t)$ of $g(t)$ against those of $f(t)$. The cross correlation of $g(t)$ and $f(t)$ can be computed as:

$$\begin{aligned} h_c(t) &= f(t) \star g(t) = \\ &\sum_{i=1}^{n-1} \left(\sum_{j=1}^{i} (f_j \star g_{n-i+j}(t - (i-1)\tau)) \right) \\ &+ \sum_{k=1}^{n} (f_k \star g_k(t - (n-1)\tau)) \\ &+ \sum_{i=n-1}^{1} \left(\sum_{j=1}^{i} (f_{n-i+j} \star g_j(t - (2n-i-1)\tau)) \right). \end{aligned}$$

The normalized cross correlation, $h(t)$, is obtained by normalizing $h_c(t)$ to $E = \sqrt{\|\sum_{i=1}^{n} f_i(t-i\tau)\|.\|\sum_{i=1}^{n} g_i(t-i\tau)\|}$. Since the segments of f and g do not overlap each other (by their definition), the energy of a signal is the sum of the energies of its components, such that $E = \sqrt{\sum_{i=1}^{n} \|f_i(t)\|. \sum_{i=1}^{n} \|g_i(t\|)}$. Finally, we expand E to obtain the final value of the normalized cross correlation between f and g as:

$$h(t) = \frac{h_c(t)}{\sqrt{\sum_{i=1}^{n} \sum_{j=1}^{n} \|f_i(t)\|.\|g_j(t)\|}}. \quad (5.2)$$

To decide on the final outcome, `VAuth` computes $\max|h(t)|$, which, according to the triangle rule, becomes:

$$\max|h(t)| \leq \max \frac{|h_c(t)|}{\sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n}\|f_i(t)\|.\|g_j(t)\|}}. \tag{5.3}$$

We assume that the segments' cross-correlation maximizes when they are aligned. That is, $max(f_i \star g_j) = \sum\limits_{t=0}^{\tau} f_i(t)g_i(t)$; otherwise, we can redefine the segments to adhere to this property. We can then separate the components of Eq. (5.3) into different components such as:

$$\begin{aligned}
\max|h(t)| &\leq \frac{1}{E}.\max(h_l, h_m, h_r), \text{where} \\
h_l &= \max_{i=1\ldots n-1}\left(\sum_{j=1}^{i}|f_j \star g_{n-i+j}(t)|\right), \\
h_m &= \max|f_k \star g_k(t)|, \text{and} \\
h_r &= \max_{i=1\ldots n-1}\left(\sum_{j=1}^{i}|f_{n-i+j} \star g_j(t)|\right).
\end{aligned} \tag{5.4}$$

The above equation describes how the final outcome is related to the results of running `VAuth` on the segments comprising $f(t)$ and $g(t)$. Two segments $f_i(t)$ and $g_j(t)$ are positively matched when their maximum of normalized cross correlation, m_{ij}, is between th and 1. Otherwise, there is a negative match.

The value of m_{ij} can be given as:

$$m_{i,j} = \max \frac{|f_i \star g_j(t)|}{\|f_i(t)\|.\|g_j(t)\|}. \tag{5.5}$$

Let $e_{i,j}$ denote the product of the energies of f_i and g_j, such that $e_{i,j} = \|f_i(t)\|.\|g_j(t)\|$. After applying the triangle rule to Eq. (5.4), the final outcome $m = \max|h(t)|$ can be given as:

$$\begin{aligned}
m \leq \frac{1}{E}.\max\Bigg(&\max_{i=1\ldots n-1}\sum_{j=1}^{i} m_{i,n-i+j}.e_{i,n-i+j}, \\
&\sum_{k=1}^{n} m_{kk}.e_{kk}, \max_{i=1\ldots n-1}\sum_{j=1}^{i} m_{n-i+j,j}.e_{n-i+j,j}.\Bigg)
\end{aligned} \tag{5.6}$$

It is evident from Eq. (5.6) how the final outcome of `VAuth` depends on computing the maximum of $2n-1$ distinct components. Each component, $\sum_{j=1}^{i} m_{ij}.e_{ij}$, is simply the weighted average of the outcomes of `VAuth` when it matches the included segments. Without loss of generality, let's consider the case when $n = 2$:

$$m \leq \max\left(\frac{m_{11}.e_{11}}{E}, \frac{m_{12}.e_{12} + m_{21}.e_{21}}{E}, \frac{m_{22}.e_{22}}{E}\right), \tag{5.7}$$

where m_{ij} are as defined in Eq. (5.5); m_{11} and e_{11} are the results of matching f_1 and g_2, m_{12} and e_{12} are those of f_1 and g_1, m_{21} and e_{21} are those of f_2 and g_2, and m_{22} and e_{22} are those of f_2 and g_1.

Given the above model of the final outcome of `VAuth` as a function of the segments composing the commands, we study the properties of `VAuth` as described below.

5.8.2 Per-segment Analysis

Eq. (5.7) reveals the importance of the per-segment analysis of `VAuth`. This step thwarts an attacker's ability to inject commands into the voice assistant system. The attacker aims to inject segments to $g(t)$ that do not bring m below th (so that `VAuth` generates a positive match according to Eq. (5.1)). If there were no per-segment analysis, an attacker could exploit matching segments to inject segments that do not match. The middle component of Eq. (5.7) explains it. Assuming that $m_{1,2}.e_{1,2}$ is large enough, the attacker can inject $g_2(t)$ such that $m_{1,2}.e_{1,2} + m_{2,1}.e_{2,1}$ is still large, despite $m_{2,1}$ being low. This happens when $e_{2,1}$ is too low, implying that the accelerometer did not record the injected segment.

The per-segment analysis of `VAuth` addresses this issue using three mechanisms. *First*, it removes all portions of $f(t)$ that fall below the running average of the noise level. These removed portions will not even be part of Eq. (5.7). So, the attacker cannot inject commands when the user is silent (no corresponding accelerometer signal).

Second, if the energy of some segment of $f(t)$ is above the noise level, VAuth normalizes its magnitude to 1, after removing the spikes. As such, it aims to make the energies of the segments of $f(t)$ uniform. The attacker cannot inject a command with very low energy as it will not be recorded by the microphone of the voice assistant. This forces $e_{2,1}$ to be comparable to $e_{1,2}$. As a result, a low value of $m_{2,1}$ reduces the value of m of Eq. (5.7). *Third*, and more importantly, The per-segment analysis of VAuth nullifies those segments which have their maximum normalized cross-correlation falling below a threshold (equal to 0.4 in our evaluation). These segments will not make it to the final decision stage, and will not be part of Eq. (5.7).

5.8.3 False Positive Rate

The results of Section 5.7 show that the false positive rate of matching is not zero for the English phonemes. Such a false positive rate opens up security holes in VAuth. We show below that while the false positive rate is not zero at the phonetic level, adding more phonemes to the command will drive the false positive rate closer to zero. In other words, the more sounds the user speaks (i.e., the longer the command is), the lower the false positive rate will be.

To show this, we will take another look at Eq. (5.6), where f_i and g_i represent the phonemes making up the command. At the phonetic level, a false positive event occurs when $m_{i,j} > th$, given that f_i does not match g_j. As evident from Eq. (5.6), when the values of $e_{i,j}$ are roughly uniform which we ensure from the per-segment analysis, the value of m is simply an average of the values of $m_{i,j}$. The final matching result $v(f, g)$, is a direct function of m. A false positive event occurs when $v(f, g) = 1$ or $m > th$, given that the underlying accelerometer ($f(t)$) and microphone ($g(t)$) signals do not match.

There are two cases available in Eq. (5.6): $i < n$ (the first and third terms in the max) and $i = n$ (the middle term in the max). The former case is simple; the final value of m is by definition a scaled-down version of the m_{ij}s. A lower value of m will lower the false positive rate.

The latter case considers n segments (phonemes) composing the command. A false positive event occurs when $m = 1/n \sum\limits_{k=1}^{n} m_{k,k} > th$, given that $f(t)$ does not match $g(t)$. In the case of phonemes false positives, one can view all $m_{i,j}$s as being drawn from the distribution $P_M(m_{i,j}) = P_M(M = m_{i,j}|f_i \neq g_j)$, where the $\neq$ operator indicates non-matching; the false positive rate is simply $P_M(M > th|f_i \neq g_j)$. The false positive rate of the whole command is then equal to $P_M(m) = P_M(\sum_{k=1}^{n} m_{k,k}/n > th|f \neq g)$. Our objective reduces to showing that $P_M(m)$ decays as n increases (i.e., more non-matching phonemes are added to the command).

The distribution $P_M(M = m_{ij})$ is an arbitrary one that only satisfies two conditions. First, it is bounded since the values of $m_{i,j}$ are limited to the interval $[0, 1]$. Second, the matching threshold, th, is larger than the mean of the distribution such that $th > E(m_{i,j})$. The empirical false positive distribution $P(m_{i,j})$ that we estimated in Section 5.7 satisfies both conditions.

We know from the Hoeffding bound that since $m_{i,j}$ are bounded, $P_M(\sum_{k=1}^{n} m_{k,k} - n.E(m) > t) \leq e^{\frac{-2t^2}{n}}$, for $t \geq 0$. Substituting $t = n.th - n.E(m)$ (which is larger than 0) yields:

$$P_M(\frac{1}{n}.\sum_{k=1}^{n} m_{nk} > th) \leq e^{-2n(th-E(m))^2}. \tag{5.8}$$

The left-hand side of Eq. (5.8) is simply the false positive rate of the command composed of n non-matching phonemes. Clearly, this false positive rate decays exponentially fast in n. Our results from the user study further confirm this analysis.

5.9 Evaluation

We now evaluate the efficacy of `VAuth` in identifying common voice assistant commands, under different scenarios and for different speakers. We demonstrate that

`VAuth` delivers almost perfect matching accuracy (True Positives, TPs) regardless of its position on the body, user accents, mobility patterns, or even across different languages. Moreover, we elaborate on the security properties of `VAuth`, demonstrating its effectiveness in thwarting various attacks. Finally, we report the delay and energy consumption of our wearable prototypes.

5.9.1 User Study

Table 5.3: The list of commands.[6]

Command	**Command**
1. How old is Neil deGrasse Tyson?	**16.** Remind me to buy coffee at 7am from Starbucks
2. What does colloquial mean?	**17.** What is my schedule for tomorrow?
3. What time is it now in Tokyo?	**18.** Where's my Amazon package?
4. Search for professional photography tips	**19.** Make a note: update my router firmware
5. Show me pictures of the Leaning Tower of Pisa	**20.** Find Florence Ion's phone number
6. Do I need an umbrella today? What's the weather like?	**21.** Show me my bills due this week
7. What is the Google stock price?	**22.** Show me my last messages.
8. What's 135 divided by 7.5?	**23.** Call Jon Smith on speakerphone
9. Search Tumblr for cat pictures	**24.** Text Susie great job on that feature yesterday
10. Open greenbot.com	**25.** Where is the nearest sushi restaurant?
11. Take a picture	**26.** Show me restaurants near my hotel
12. Open Spotify	**27.** Play some music
13. Turn on Bluetooth	**28.** What's this song?
14. What's the tip for 123 dollars?	**29.** Did the Giants win today?
15. Set an alarm for 6:30 am	**30.** How do you say good night in Japanese?

To support the conclusions derived from our model, we conducted a detailed user study of the `VAuth` prototype with 18 users and under 6 different scenarios. We tested how `VAuth` performs at three positions, each corresponding to a different form of wearable (Fig. 5.4) eyeglasses, earbuds, and necklace. At each position, we tested two cases, asking the user to either stand still or jog. In each scenario, We asked the participants to speak 30 phrases/commands (listed in Table 5.3). These phrases represent common commands issued to the "Google Now" voice assistant. In what follows, we report `VAuth`'s detection accuracy (TPs) and false positives (FPs) when doing a pairwise matching of the commands for each participant. We collected no

[6]inspired from: `http://www.greenbot.com/article/2359684/android/a-list-of-all-the-ok-google-voice-commands.html`

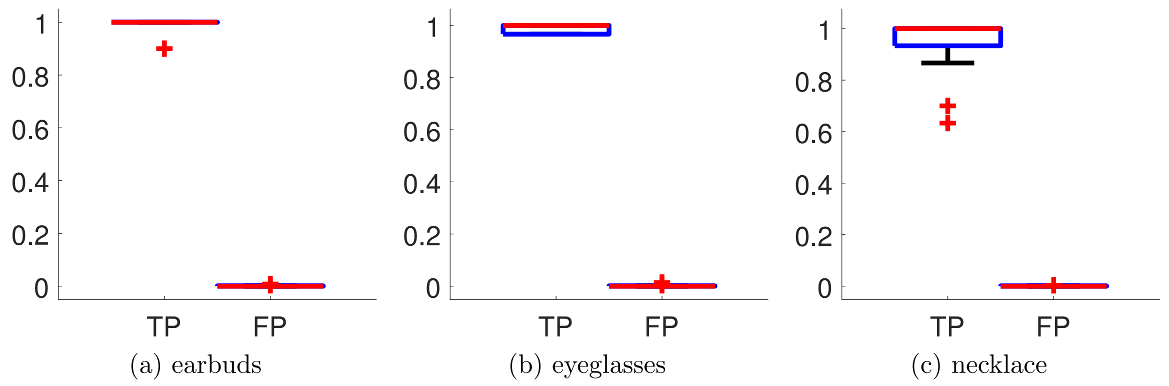


Figure 5.11: The detection accuracy of `VAuth` for the 18 users in the still position.

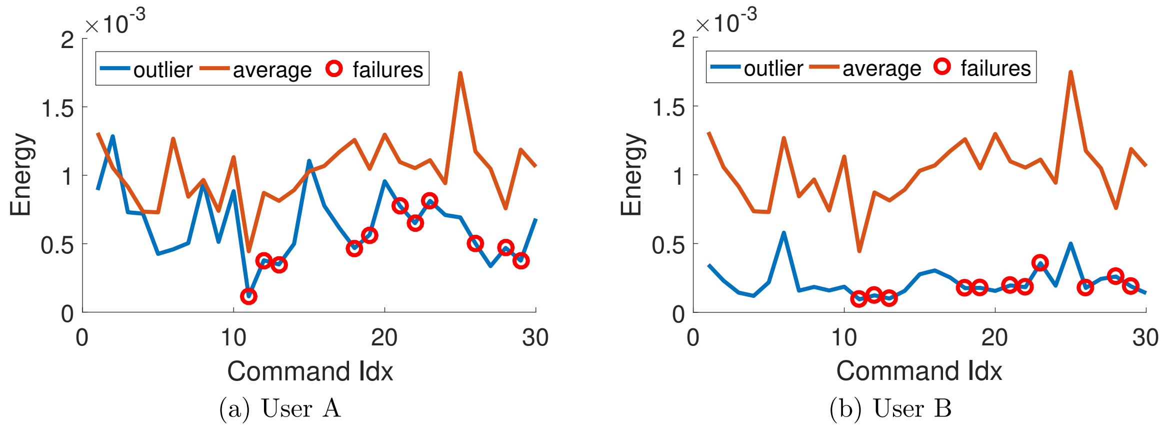


Figure 5.12: The energy levels of the outlier users (in Fig. 5.11c) compared to average users. The circles represent commands of the outlier users that `VAuth` fails to match.

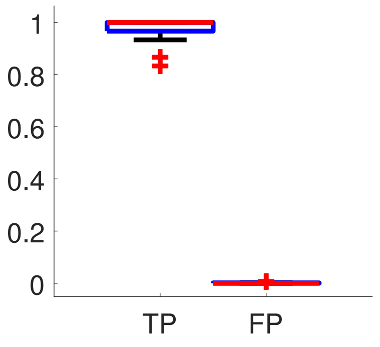
personally identifiable information from the individuals, and the data collection was limited to our set of commands and posed no privacy risk to the participants. As such, our user study meets the IRB exemption requirements of our institution.

Still `VAuth` delivers high detection accuracy (TPs), with the overall accuracy rate very close to 100% (more than 97% on average). This indicates most of the commands are correctly authenticated from the first trial and `VAuth` does not introduce a usability burden to the user. The false positive rate is 0.09% on average, suggesting that very few signals will leak through our authentication. These false positive events occur because the per-segment analysis of our matching algorithm removes all non-matching segments from both signals, which ensures the security properties of `VAuth`. In these cases, when the remaining segments for the microphone signal accidentally match what the user said and leak through `VAuth`, the voice recognition system (Voice-to-Text) fails to pick them up as sensible voice commands. Fig. 5.11 shows the overall distribution of detection results for each scenario.

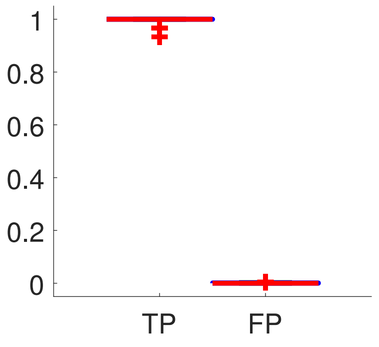
`VAuth` performs almost perfect in two wearable scenarios, eyeglasses and earbuds, but has two outliers regarding the detection accuracy in the case of the necklace. We looked into the commands that `VAuth` fails to recognize and found they happen when there are significant energy dips in the voice level. Fig. 5.12 reports the energy levels of the voice sessions for our two outlier users compared to the average across users. This suggests both participants used a lower (than average) voice when doing the experiments which did not generate enough energy to ensure the authentication.

Mobility We asked the participants to repeat the experiments at each position while jogging. Our algorithm successfully filters the disturbances introduced by moving, breathing and `VAuth`'s match accuracy remains unaffected (see Fig. 5.13). In fact, we noticed in certain cases, such as for the two outliers observed in our previous experiments, the results are even better. We studied the difference between their samples in the two scenarios and found both accelerometer and microphone received significantly higher energy in the jogging scenario even after we filtered out the signals introduced by movement. One explanation is users are aware of the disturbance introduced by jogging and try to use louder voice to compensate. This observation is consistent across most of our participants, not just limited to the two outliers.

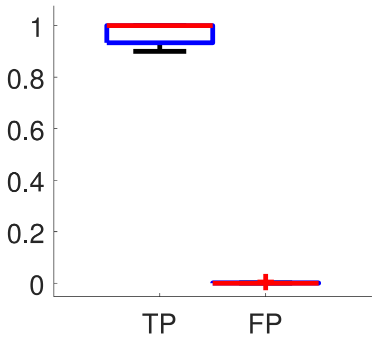
Language We translated the list of 30 commands into four other languages — Arabic, Chinese, Korean and Persian — and recruited four native speakers of these languages. We asked the participants to place and use `VAuth` at the same three positions. As shown in Fig. 5.14, `VAuth` performs surprisingly well, even though the



(a) earbuds

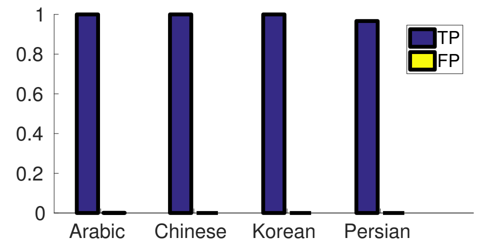


(b) eyeglasses

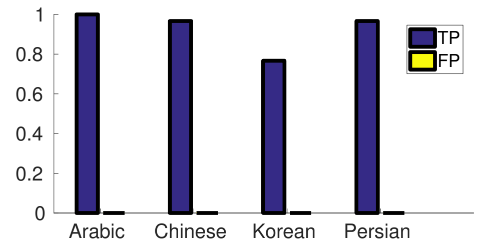


(c) necklace

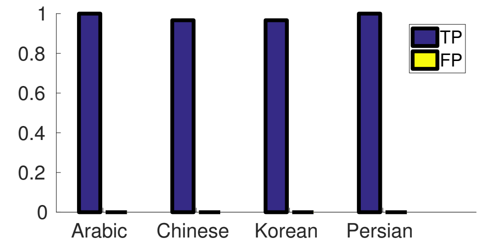
Figure 5.13: The detection accuracy of `VAuth` for the 18 users in the moving position.



(a) earbuds



(b) eyeglasses



(c) necklace

Figure 5.14: The detection accuracy of `VAuth` for the 4 different languages.

`VAuth` prototype was trained on English phonemes (Section 5.6.3). `VAuth` delivers almost perfect detection accuracy, except for one case, with the user speaking Korean when wearing eyeglasses. The Korean language lacks nasal consonants, and thus does not generate enough vibrations through the nasal bone [113].

5.9.2 Security Properties

In Section 5.4, we listed three types of adversaries against which we aim to protect the voice assistant systems. `VAuth` can successfully thwart attacks by these adversaries

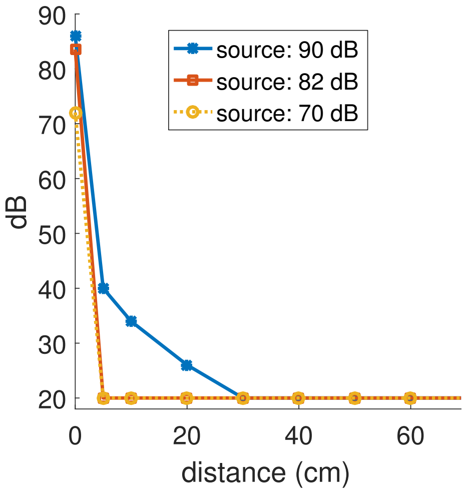
Table 5.4: The protections offered by `VAuth`.

Scenario	**Adversary**	**Example**	**Silent User**	**Speaking User**
A	Stealthy	mangled voice, wireless-based	✔	✔
B	Biometric Override	replay, user impersonation	✔	✔
C	Acoustic Injection	direct communication, loud voice	distance cut-off	distance cut-off

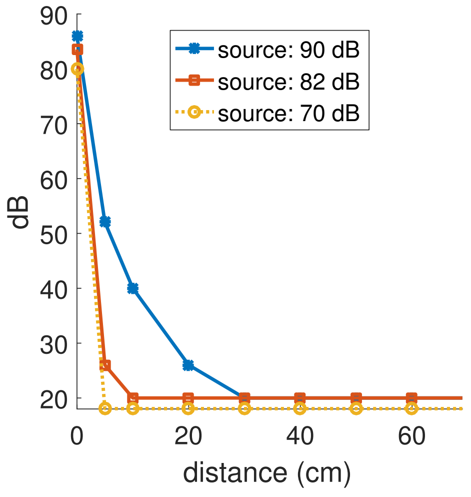
through its multi-stage matching algorithm. Table 5.4 lists the protections offered by `VAuth` when the user is silent and actively speaking. Here, we use the evaluation results in Section 5.9 to elaborate on `VAuth`'s security features for each attack scenario and both cases when the user is silent and speaking.

Silent User When the user is silent, `VAuth` completely prevents any unauthorized access to the voice assistant. In Section 5.7.3, we evaluate the false positive rate of `VAuth` mistakenly classifying noise while the user is silent for all English phonemes. We show that `VAuth` has a zero false positive rate. When the user is silent, the adversary *cannot inject* any command for the voice assistant, especially for scenarios A and B of Section 5.4. There is an exception, however, for scenario C; an adversary can employ a very loud sound to induce vibrations at the accelerometer chip of `VAuth`. Note that, since the accelerometer only senses vibrations at the z-axis, the attacker must make the extra effort to direct the sound wave perpendicular to the accelerometer sensing surface. Next, we will show that beyond a cut-off distance of 30cm, very loud sounds (directed at the z-axis of the accelerometer) do not induce accelerometer vibrations. Therefore, to attack `VAuth`, an adversary has to play a very loud sound within less than an arm's length from the user's body — which is highly improbable.

We conduct experiments on the cut-off distances in two scenarios: the first with `VAuth` exposed and the second with `VAuth` covered with cotton clothing. Fig. 5.15



(a) Exposed accelerometer



(b) Covered accelerometer

Figure 5.15: The magnitude of the sensed over-the-air vibrations by the accelerometer as a function of the distance between the sound source and the accelerometer.

reports how the accelerometer chip of `VAuth` reacts to over-the-air sound signals of different magnitudes at different distances. In each of these scenarios, we played a white noise at three sound levels:[7] 2x, 4x and 8x the conversation level at 70dB, 82db and 90dB, respectively. The noise is directed perpendicularly to the sensing surface of the accelerometer. Fig. 5.15a shows the recorded magnitude of the accelerometer signal as a function of the distance between the sound source and `VAuth` when it is exposed. As evident from the plots, there is a cut-off distance of 30cm, where `VAuth`'s accelerometer cannot sense even the loudest of the three sound sources. For the other two sounds, the cut-off distance is 5cm. Beyond the cut-off distance, the magnitude of the recorded signal is the same as that in a silent scenario. This indicates that an adversary cannot inject commands with a high sound level beyond some cut-off distance. These results are consistent with the case of `VAuth` covered with cotton, as shown in Fig. 5.15b. The cut-off is still 30cm for the loudest sound. It is worth noting that the recorded signal at the microphone does not change magnitude as drastically as a function of the distance. At a distance of 1m from the sound source, the audio

[7]`http://www.industrialnoisecontrol.com/comparative-noise-examples.htm`

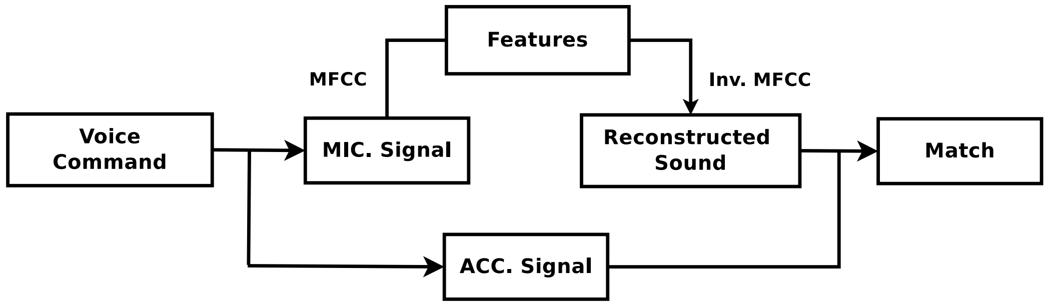


Figure 5.16: The flow of the mangling voice analysis.

signal loses at most 15dB of magnitude.

Speaking User On the other hand, the adversary may try to launch an attack on the voice assistant system when the user is actively speaking. Next, we show how `VAuth` can successfully thwart the stealthy attacks in scenario A. We will show, in the most extreme case scenario, how `VAuth` can completely distinguish the accelerometer samples of the voice spoken by the user from the reconstructed sound of the same command, even when the reconstructed voice sounds the same to the human listener as the original one.

Vaidya *et al.* [19, 104] presented an attack that exploits the gap between voice recognition system and human voice perception. It constructs mangled voice segments that match the MFCC features of an injected voice command. An ASR engine can recognize the command, but not the human listener. This and similar attacks rely on performing a search in the MFCC algorithm parameter space to find voice commands that satisfy the above feature.

Performing an exhaustive search on the entire parameter space of the MFCC generation algorithm is prohibitive. Therefore, to evaluate the effectiveness of `VAuth` against such an attack, we consider its worst-case scenario. Fig. 5.16 shows the evaluation flow. For each of the recorded command of the previous section, we extract the MFCCs for the full signal and use them to reconstruct the voice signal. Finally, we execute `VAuth` over the reconstructed voice segment and the corresponding accelerometer sample to test for a match.

We fixed the MFCC parameters as follows: 256 samples for the hop time, 512 samples for the window time, and 77 as the length of the output coefficient vector. We vary the number Mel filter bands between 15 and 30. At 15 Mel filter bands, the reconstructed voice command is similar to what is reported in existing attacks [104]. At 30 Mel filter bands, the reconstructed voice command is very close to the original; it shares the same MFCCs and is easily identifiable when played back.

The question that we aim to address is whether reducing the sound signal to a set of features and reconstructing back the original signal preserves all the acoustic features needed for `VAuth` to perform a successful matching with the corresponding accelerometer signal. If not, then the reconstructed sound will not even match the voice it originated from. Therefore, any mangled voice will not match the user's speech as measured by `VAuth`, so that `VAuth` could successfully thwart the attack.

In all cases, while the original microphone signal matches accelerometer signals near perfectly as indicated before, the reconstructed sound failed to match the accelerometer signal in 99% of the evaluated cases. Of 3240 comparisons (2 Mel filter band lengths per command, 90 commands per user and 18 users), the reconstructed sound matched only a handful of accelerometer samples, and only in cases where we used 30 Mel filter bands. Indeed, those sound segments were very close to the original sound segment that corresponds to the matched accelerometer samples. To constitute an attack, the mangled voice segment is not supposed to originate from the sound the user is speaking, let alone preserving discernible acoustic features. This demonstrates that `VAuth` matches the time-domain signals in their entirety, thwarting such attacks on the voice assistant and recognition systems.

Last but not least, we tested `VAuth` with the set of mangled voice commands[8] used by Carlini *et al.* [19]. We asked four different individuals to repeat these commands

[8]http://www.hiddenvoicecommands.com/

while wearing `VAuth`. The accelerometer samples corresponding to each command do not match their mangled voice counterparts.

In scenario B, an attacker also fails to overcome `VAuth`'s protection. We indicated earlier in Section 5.7.2 and in this section that `VAuth` successfully distinguishes the phonemes and commands of the same user. We further confirm that `VAuth` can differentiate the same phoneme or command across different users. Moreover, even if the user is speaking and the adversary is replaying another sound clip of the same user, `VAuth` can differentiate between the microphone and accelerometer samples and stop the attack. Finally, `VAuth` might result in some false positives (albeit very low). As explained earlier, these false positive take place because the remaining segments after the per-segment stage of `VAuth` match, and thus do not represent a viable attack vector. It is worth noting that `VAuth` could use a more stringent classifier that is tuned to force the false positive rate to be 0. This will come at the cost of usability but could be preferable in high-security situations.

5.9.3 Delay and Energy

We measure the delay experienced at the voice assistant side and the energy consumption of the wearable component, using our prototype. As shown in Fig. 5.2, `VAuth` incurs delay only during the matching phase: when `VAuth` uploads the accelerometer and microphone signals to the remote service and waits for a response. According to our test on the same list of 30 commands, we found that a successful match takes 300–830ms, with an average of 364ms, while an unsuccessful match takes 230–760ms, with an average of 319ms. The response time increases proportionally to the length of the commands, but matching a command containing more than 30 words still takes less than 1 second. We expect the delay to decrease further if switching from our Matlab-based server to a full-fledged web server.

When the wearable component transmits accelerometer signals, it switches be-

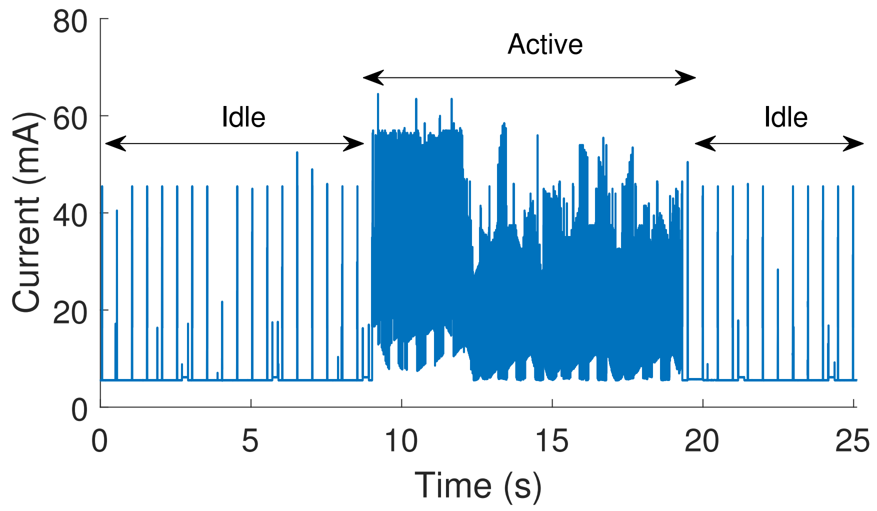


Figure 5.17: Current levels of the prototype in the idle and active states.

tween two states: idle state that keeps the connection alive and active state that actually transmits the data. We connected our prototype to the Monsoon power monitor and recorded the current levels of the prototype in these two states when powered by a fixed voltage (4V). Fig. 5.17 illustrates the changes of the current levels when our prototype switches from idle to active and then back to idle. We observed that under active state, our prototype consumes as much as 31mA, while under idle state, it only consumes an average of 6mA. Most of the energy is used to keep the Bluetooth connection and transmit data (in the active state) — the energy consumed by the accelerometer sensor is almost negligible.

Assuming the user always keeps the wearable open at daytime and sends 100 voice commands per day (each voice command takes 10 seconds). Our prototype consumes 6.3mA on average. This might even be an overestimation since 90% of the users issue voice commands at most once per day according to our survey. A typical 500mAh Li-Ion battery used by wearables (comparable to a US quarter coin) can power our prototype for around a week. 80% of the participants in our usability survey think they have no problem with recharging the wearable on a weekly basis. We conducted all the analyses on our prototype which directly utilizes off-the-shelf hardware chips without any optimization, assuming that VAuth is provided as a standalone wearable. If incorporated into an existing wearable device, VAuth will only introduce an additional energy overhead of less than 10mAh per day.

5.10 Discussion

In our prototype implementation, we enforce the same policy for all voice commands: if the authentication passes, execute else drop the command. However, one can implement customized policy for different commands. For example, some commands, such as time/weather inquiry, are not privacy/security-sensitive, so VAuth can execute them directly without going through additional authentication process; other commands, such as controlling home appliances might be highly sensitive, and hence VAuth should promptly warn the user instead of simply dropping the command. This can be implemented by extending the Intent interception logic in our prototype implementation, making VAuth react differently according to different Intent actions, data, and types.

Besides its excellent security properties, VAuth has a distinct advantage over existing technologies — it is wear-and-use without any user-specific, scenario-dependent training. Although we used machine learning to facilitate matching decision in our algorithm, we only trained once (on English phonemes of a test user) and then applied it in all other cases. Our evaluation demonstrates that VAuth is robust to changes in accents, speed of speech, mobility, or even languages. This significantly increases the usability of VAuth.

5.11 Conclusion

In this chapter, we have proposed VAuth, a system that provides continuous authentication for voice assistants. We demonstrated that even though the accelerometer information collected from the facial/neck/chest surfaces might be weak, it contains enough information to correlate it with the data received via microphone. `VAuth` provides extra physical assurance for voice assistant users and is an effective measure against various attack scenarios. It avoids the pitfalls of existing voice authentication mechanisms. Our evaluation with real users under practical settings shows high accuracy and very low false positive rate, highlighting the effectiveness of `VAuth`. In future, we would like to explore more configurations of `VAuth` that will promote wider real-world deployment and adoption.

CHAPTER VI

Conclusions And Future Works

As mobile becomes the central theme of our digital consumer market, new techniques have been continuously invented for, and introduced to the mobile platform. In this dissertation, we have demonstrated the risks when these techniques are directly imported and applied without being aware of the trust relationship in the mobile ecosystem. Specifically, we focused on various communication interfaces that enable the information and control flow of the current mobile ecosystem.

6.1 Conclusions

In Chapter II, we conducted an in-depth analysis on a client-side IPC interface, Android Binder. According to our analysis of more than 100 vulnerabilities on this surface, we discovered a common misconception of where the security boundary is for Android system services — many may assume the security/trust boundary is at the client-side public APIs, and whatever happens thereafter is free from obstruction since it is already in the system zone. This misconception is understandable since Android provides the convenient abstraction of AIDL and automatically generates codes in the IPC stack. In other words, even though Android system services depend heavily on IPC, the developers are likely to be agnostic of that. We addressed this problem with *BinderCracker*, a precautionary testing framework that achieves automatic discovery of vulnerabilities. It supports context-aware fuzzing, automatically resolves dependency across IPC transactions, and achieves 7x better effectiveness than simple black-box fuzzing. We also highlighted the urgent need for attack attribution for IPC-based attacks. Due to the lack of transparency, no one knows why a system service crashed and who caused it. We proposed proper OS-level support by building an OS-level runtime diagnostics tool on the binder surface. We believe it will be essential for any in-depth runtime analysis. Our solutions are useful, practical, and easily integratable in the development/deployment cycle of Android.

In Chapters III and IV, we demonstrated how to restrict the potential information flow between apps by monitoring, characterizing and reducing the underlying linkability across them. In Chapter III, we proposed the concept of DLG (Dynamic Linkability Graph), which captures both the quality and quantity of the linkability across apps. This allows us to measure the potential threat of unregulated aggregation during runtime and promptly warn users of the associated risks. We observed how real-world apps abuse OS-level information and IPCs to establish linkability, and proposed a practical countermeasure, `LinkDroid`. It provides runtime monitoring and mediation of linkability across apps, introducing a new dimension to privacy protection on mobile device. Our evaluation with real users has shown that `LinkDroid` is effective in reducing the linkability across apps and only incurs marginal overhead. In Chapter IV, we further discussed an alternative solution, called `Mask`, in the user level for the same problem of unregulated aggregation. It introduces a set of *private execution modes* and give users options to maintain multiple isolated profiles of the same app. `Mask` achieves this by enforcing user-level sandboxes and exploiting the dynamic linking process in Android. All user behaviors originate from the mobile apps, either directly or indirectly. By enabling the proposed private execution modes which isolate app usages at this very source, `Mask` provides a client-side solution without requiring any change to the existing ecosystem.

In Chapter V, we secured an increasingly popular human-machine interaction channel, the voice channel, for mobile voice assistants. We proposed `VAuth`, a system that provides continuous authentication for voice assistants. We demonstrated that even though the accelerometer information collected from the facial/neck/chest surface might be weak, it contains enough information to correlate it with the data collected from microphone. `VAuth` provides extra physical assurance for voice assistant users, effectively thwarts mangling voice attacks and successfully blocks unauthenticated voice commands replayed by an attacker or impersonated by other users, It avoids the pitfalls of existing voice authentication mechanisms and has a distinct advantage over previous technologies — it is wear-and-use without any user-specific, scenario-dependent training. Our evaluation with real users under practical settings shows high accuracy and very low false positive rate, regardless of variations in accents, mobility patterns (still vs. jogging), or even across languages (Arabic, Chinese, English, Korean, Persian).

6.2 Future Directions

In this thesis, we elaborated on the challenges encountered in an emerging mobile ecosystem. Many of our experiences can be reused when dealing with other interfaces in the mobile ecosystem or similar interfaces in other domains, such as vehicular networks and Internet of Things (IoTs). This opens up new directions for future explorations.

Apps make use of various sensory data provided by the mobile OS. However, most of the apps naively assume that the sensory data received is authentic. This may cause severe privacy and security problems, not only to the users who are currently using the app, but also to businesses running the app. For example, the location information is utilized by many important/sensitive apps or services, for social, search or authentication purposes. However, it is very easy to fake location data in current mobile OSes and attackers can utilize this loophole to spam nearby users, exploit mobile games (e.g., Pokemon GO), and commit frauds. Similar exploitations are prevalent in the current mobile ecosystem and we believe this is an interesting problem worthy of further exploration.

Many of the problems we discussed in this thesis also exist in other domains and deserve more attention. For example, in-vehicle communications work on top of the CAN (Controller Area Network) bus. Overwhelmed by the unprecedented trend of Internet-powered in-vehicle applications, the CAN protocol is significantly outdated in terms of security or privacy. Essentially, the CAN protocol provides neither built-in security nor (serious) sanity checks on the messages transmitted through the network. It has been demonstrated that any component with access to the CAN bus can introduce severe physical security issues, putting the driver's and passengers' lives in danger [21]. This calls for the design and implementation of precautionary testing, as well as runtime diagnostic tools on the CAN interface. The vehicle should at least be able to trace attacks to specific ECU (Electronic Control Unit). Another potential direction is introducing linkability tracking/control to IoT devices. IoT devices serving each household and each person are predicted to grow very fast and soon become untraceable. All the devices will be connected to Internet and uploading private data that can be attributed to users. It will be essential to ensure that information from multiple IoT devices should not be linkable unless there are legitimate needs, and users should be given the options to opt out. Without a proper scheme of enforcing linkability, our life will become transparent eventually, and probably in an unwanted way.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Forecast: Mobile app stores, worldwide, 2013 update. `http://www.gartner.com/DisplayDocument?id=2584918`.

[2] 2013: a look back at the year in acquisitions. `http://vator.tv/news/2013-12-07-2013-a-look-back-at-the-year-in-acquisitions`.

[3] G. Aggarwal, E. Bursztein, C. Jackson, and D. Boneh. An analysis of private browsing modes in modern browsers. In *Proceedings of the 19th USENIX conference on Security*, pages 6–6. USENIX Association, 2010.

[4] D. Amalfitano, A. R. Fasolino, and P. Tramontana. A gui crawling-based technique for android mobile application testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 252–261. IEEE, 2011.

[5] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 258–261. ACM, 2012.

[6] Androbench. `http://www.androbench.org/wiki/AndroBench`.

[7] Android interface definition language (aidl). `http://developer.android.com/guide/components/aidl.html`.

[8] Angry birds and 'leaky' phone apps targeted by nsa and gchq for user data. `http://www.theguardian.com/world/2014/jan/27/nsa-gchq-smartphone-app-angry-birds-personal-data`.

[9] A. J. Aviv, B. Sapp, M. Blaze, and J. M. Smith. Practicality of accelerometer side channels on smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 41–50. ACM, 2012.

[10] M. Backes, A. Kate, M. Maffei, and K. Pecina. Obliviad: Provably secure and practical online behavioral advertising. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 257–271, Washington, DC, USA, 2012. IEEE Computer Society.

[11] M. Balakrishnan, I. Mohomed, and V. Ramasubramanian. Where's that phone?: geolocating ip addresses on 3g networks. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 294–300. ACM, 2009.

[12] M. Baloul, E. Cherrier, and C. Rosenberger. Challenge-based speaker recognition for mobile authentication. In *Biometrics Special Interest Group (BIOSIG), 2012 BIOSIG-Proceedings of the International Conference of the*, pages 1–7. IEEE, 2012.

[13] A. Bamis and A. Savvides. Lightweight Extraction of Frequent Spatio-Temporal Activities from GPS Traces. In *IEEE Real-Time Systems Symposium*, pages 281–291, 2010.

[14] Y. Ben-Itzhak. What if smart devices could be hacked with just a voice? `http://now.avg.com/voice-hacking-devices/`, Sep. 2014.

[15] Bluetooth SIG. Specification of the Bluetooth System. Version 4.2, Dec. 2014. `https://www.bluetooth.org/en-us/specification/adopted-specifications`.

[16] P. Boersma. Accurate short-term analysis of the fundamental frequency and the harmonics-to-noise ratio of a sampled sound. *Institute of Phonetic Sciences - University of Amsterdam*, 17:97–110, 1993.

[17] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks.

[18] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *Presented as part of the 22nd USENIX Security Symposium*, pages 131–146, Berkeley, CA, 2013. USENIX.

[19] N. Carlini, P. Mishra, T. Vaidya, Y. Zhang, M. Sherr, C. Shields, D. Wagner, and W. Zhou. Hidden voice commands. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 513–530, Austin, TX, Aug. 2016. USENIX Association.

[20] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[21] K.-T. Cho and K. G. Shin. Fingerprinting electronic control units for vehicle intrusion detection. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 911–927, Austin, TX, Aug. 2016. USENIX Association.

[22] C. Cornelius, Z. Marois, J. Sorber, R. Peterson, S. Mare, and D. Kotz. Vocal resonance as a passive biometric. 2014.

[23] C. Cornelius, R. Peterson, J. Skinner, R. Halter, and D. Kotz. A wearable system that knows who wears it. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 55–67, New York, NY, USA, 2014. ACM.

[24] Cve-2015-1474. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1474`.

[25] Cve-2015-1528. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1528`.

[26] Cve-2015-6612. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6612`.

[27] Cve-2015-6620. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6620`.

[28] A. Das, O. K. Manyam, M. Tapaswi, and V. Taranalli. Multilingual spoken-password based user authentication in emerging economies using cellular phone networks. In *Spoken Language Technology Workshop, 2008. SLT 2008. IEEE*, pages 5–8. IEEE, 2008.

[29] D. Davidson and B. Livshits. Morepriv: Mobile os support for application personalization and privacy. Technical report, MSR-TR, 2012.

[30] 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 *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones &#38; Mobile Devices*, SPSM '14, pages 63–74, New York, NY, USA, 2014. ACM.

[31] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In *NDSS*, 2011.

[32] K. O. Elish, D. Yao, and B. G. Ryder. On the need of precise inter-app icc classification for detecting android malware collusions. In *Proceedings of IEEE Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy*, 2015.

[33] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, volume 10, pages 255–270, 2010.

[34] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[35] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.

[36] K. Fawaz and K. G. Shin. Location privacy protection for smartphone users. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 239–250. ACM, 2014.

[37] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[38] A. P. Felt, S. Egelman, M. Finifter, D. Akhawe, and D. Wagner. How to ask for permission. In *Proceedings of the 7th USENIX Conference on Hot Topics in Security*, HotSec'12, pages 7–7, Berkeley, CA, USA, 2012. USENIX Association.

[39] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, pages 3:1–3:14, New York, NY, USA, 2012. ACM.

[40] H. Feng, K. Fawaz, and K. G. Shin. Linkdroid: reducing unregulated aggregation of app usage behaviors. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 769–783, 2015.

[41] Fuzzing android system services by binder call. `https://www.blackhat.com/docs/us-15/materials/us-15-Gong-Fuzzing-Android-System-Services-By-Binder-Call-To-Escalate-Privilege.pdf`.

[42] P. Golle and K. Partridge. On the anonymity of home/work location pairs. In *Proceedings of Pervasive '09*, pages 390–397, Berlin, Heidelberg, 2009. Springer-Verlag.

[43] Google says there are now 1.4 billion active android devices worldwide. `http://www.androidcentral.com/google-says-there-are-now-14-billion-active-android-devices-worldwide`.

[44] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[45] S. Han, J. Jung, and D. Wetherall. A study of third-party tracking by mobile apps in the wild. Technical report, UW-CSE, 2011.

[46] Haskins Laboratories. The Acoustic Theory of Speech Production: the source-filter model. `http://www.haskins.yale.edu/featured/heads/mmsp/acoustic.html`.

[47] Hey your parcel looks bad. `https://www.blackhat.com/docs/asia-16/materials/asia-16-He-Hey-Your-Parcel-Looks-Bad-Fuzzing-And-Exploiting-Parcelization-Vulnerabilities-In-Android.pdf`.

[48] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 639–652, New York, NY, USA, 2011. ACM.

[49] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and Communications Security*, pages 639–652. ACM, 2011.

[50] How mobile apps stack up against mobile browsers. `http://www.emarketer.com/Article/How-Mobile-Apps-Stack-Up-Against-Mobile-Browsers/1013462`.

[51] C. Hu and I. Neamtiu. Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, pages 77–83. ACM, 2011.

[52] Improving your code with lint. `http://developer.android.com/tools/debugging/improving-w-lint.html`.

[53] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. android and mr. hide: fine-grained permissions in android applications. In *Proceedings of Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 3–14. ACM, 2012.

[54] D. Kantola, E. Chin, W. He, and D. Wagner. Reducing attack surfaces for intra-application communication in android. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 69–80, New York, NY, USA, 2012. ACM.

[55] C. Kasmi and J. Lopes Esteves. Iemi threats for information security: Remote command injection on modern smartphones. *Electromagnetic Compatibility, IEEE Transactions on*, 57(6):1752–1755, 2015.

[56] J. Krumm. Inference attacks on location tracks. In *Proceedings of the 5th international conference on Pervasive computing*, PERVASIVE'07, pages 127–143, Berlin, Heidelberg, 2007. Springer-Verlag.

[57] M. Kunz, K. Kasper, H. Reininger, M. Möbius, and J. Ohms. Continuous speaker verification in realtime. In *BIOSIG*, pages 79–88, 2011.

[58] S. Labitzke, I. Taranu, and H. Hartenstein. What your friends tell others about you: Low cost linkability of social network profiles. In *Proc. 5th International ACM Workshop on Social Network Mining and Analysis, San Diego, CA, USA*, 2011.

[59] B. Lee, L. Lu, T. Wang, T. Kim, and W. Lee. From zygote to morula: Fortifying weakened aslr on android. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 424–439, Washington, DC, USA, 2014. IEEE Computer Society.

[60] S. Lee, E. L. Wong, D. Goel, M. Dahlin, and V. Shmatikov. πbox: a platform for privacy-preserving apps. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, pages 501–514. USENIX Association, 2013.

[61] M. Liberman. Linguistics 001 – sound structure of language. `http://www.ling.upenn.edu/courses/ling001/phonology.html`. Accessed: 2016-05-23.

[62] Y.-A. S. Lien, C. R. Calabrese, C. M. Michener, E. H. Murray, J. H. Van Stan, D. D. Mehta, R. E. Hillman, J. P. Noordzij, and C. E. Stepp. Voice relative fundamental frequency via neck-skin acceleration in individuals with voice disorders. *Journal of Speech, Language, and Hearing Research*, 58(5):1482–1487, 2015.

[63] M. B. Lindasalwa Muda and I. Elamvazuthi. Voice recognition algorithms using mel frequency cepstral coefficient (mfcc) and dynamic time warping (dtw) techniques. *Journal Of Computing*, 2(3):138–143, 2010.

[64] A. M. Lund. Measuring usability with the USE questionnaire. *Usability Interface*, 8:3–6, 2001.

[65] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[66] A. K. Maji, F. A. Arshad, S. Bagchi, and J. S. Rellermeyer. An empirical study of the robustness of inter-component communication in android. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[67] Majority of digital media consumption now takes place in mobile apps. `https://techcrunch.com/2014/08/21/majority-of-digital-media-consumption-now-takes-place-in-mobile-apps/`.

[68] S. Mare, A. M. Markham, C. Cornelius, R. Peterson, and D. Kotz. Zebra: zero-effort bilateral recurring authentication. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 705–720. IEEE, 2014.

[69] C. Marforio, A. Francillon, S. Capkun, S. Capkun, and S. Capkun. *Application collusion attack on the permission-based security model and its implications for modern smartphone systems*. Department of Computer Science, ETH Zurich, 2011.

[70] P. Marquardt, A. Verma, H. Carter, and P. Traynor. (sp) iphone: decoding vibrations from nearby keyboards using mobile phone accelerometers. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 551–562. ACM, 2011.

[71] A. Martin and G. Hugo. Banking biometrics: hacking into your account is easier than you think. `https://www.ft.com/content/959b64fe-9f66-11e6-891e-abe238dee8e2`.

[72] R. Martin. Listen Up: Your AI Assistant Goes Crazy For NPR Too. `http://www.npr.org/2016/03/06/469383361/listen-up-your-ai-assistant-goes-crazy-for-npr-too`, Mar. 2016.

[73] T. McDonnell, B. Ray, and M. Kim. An empirical study of api stability and adoption in the android ecosystem. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 70–79. IEEE, 2013.

[74] D. D. Mehta, J. H. V. Stan, and R. E. Hillman. Relationships between vocal function measures derived from an acoustic microphone and a subglottal neck-surface accelerometer. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 24(4):659–668, April 2016.

[75] D. D. Mehta, M. Zañartu, S. W. Feng, H. A. Cheyne, and R. E. Hillman. Mobile voice health monitoring using a wearable accelerometer sensor and a smartphone platform. *Biomedical Engineering, IEEE Transactions on*, 59(11):3090–3096, 2012.

[76] R. Metz. Voice Recognition for the Internet of Things. `https://www.technologyreview.com/s/531936/voice-recognition-for-the-internet-of-things/`, Oct. 2014.

[77] M. F. Mokbel, C.-Y. Chow, and W. G. Aref. The new casper: query processing for location services without compromising privacy. In *Proceedings of the 32nd international conference on Very large data bases*, pages 763–774. VLDB Endowment, 2006.

[78] L. Myers. An Exploration of Voice Biometrics. `https://www.sans.org/reading-room/whitepapers/\\authentication/exploration-voice-biometrics-1436`, April 2004.

[79] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 111–125. IEEE, 2008.

[80] M. Nauman, S. Khan, and X. Zhang. Apex: Extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 328–332, New York, NY, USA, 2010. ACM.

[81] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 328–332. ACM, 2010.

[82] Nuance Cloud Services. Http services 1.0 programmer's guide. https://developer.nuance.com/public/Help/HttpInterface /HTTP_web_services_for_NCS_clients_1.0_ programmer_s_guide.pdf, Dec. 2013.

[83] Number of apps available in leading app stores as of july 2015. `http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/`.

[84] E. Owusu, J. Han, S. Das, A. Perrig, and J. Zhang. Accessory: password inference using accelerometers on smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, page 9. ACM, 2012.

[85] S. Panjwani and A. Prakash. Crowdsourcing attacks on biometric systems. In *Tenth Symposium on Usable Privacy and Security, SOUPS 2014, Menlo Park, CA, USA, July 9-11, 2014*, pages 257–269, 2014.

[86] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 71–72. ACM, 2012.

[87] O. Peles and R. Hay. One class to rule them all: 0-day deserialization vulnerabilities in android. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[88] G. Petracca, Y. Sun, T. Jaeger, and A. Atamli. Audroid: Preventing attacks on audio channels in mobile devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 181–190, New York, NY, USA, 2015. ACM.

[89] J. Platt. Fast training of support vector machines using sequential minimal optimization. In *Advances in Kernel Methods - Support Vector Learning.* MIT Press, January 1998.

[90] M. Rangwala, P. Zhang, X. Zou, and F. Li. A taxonomy of privilege escalation attacks in android applications. *Int. J. Secur. Netw.*, 9(1):40–55, Feb. 2014.

[91] R. Sasnauskas and J. Regehr. Intent fuzzer: Crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, WODA+PERTEA 2014, pages 1–5, New York, NY, USA, 2014. ACM.

[92] Security updates and resources. `https://source.android.com/security/overview/updates-resources.html`.

[93] A. Shabtai, Y. Fledel, and Y. Elovici. Securing android-powered mobile devices using selinux. *IEEE Security & Privacy*, (3):36–44, 2009.

[94] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google android: A comprehensive security assessment. *IEEE Security and Privacy*, 8(2):35–44, Mar. 2010.

[95] S. Shekhar, M. Dietz, and D. S. Wallach. Adsplit: separating smartphone advertising from applications. In *Proceedings of the 21st USENIX conference on Security symposium*, pages 28–28. USENIX Association, 2012.

[96] S. Smalley and R. Craig. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS*, volume 310, pages 20–38, 2013.

[97] Smartphone os market share, q4 2014. `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`.

[98] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating user privacy in android ad libraries. *IEEE Mobile Security Technologies (MoST)*, 2012.

[99] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS*, 2015.

[100] S. Tambe, N. Vedagiri, N. Abbas, and J. E. Cook. Ddl: Extending dynamic linking for program customization, analysis, and evolution. In *In Proc. International Conference on Software Maintenance*, 2005.

[101] O. Tripp and J. Rubin. A bayesian approach to privacy enforcement in smartphones. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 175–190, Berkeley, CA, USA, 2014. USENIX Association.

[102] T. Trippel, O. Weisse, W. Xu, P. Honeyman, and K. Fu. WALNUT: Waging doubt on the integrity of mems accelerometers with acoustic injection attacks. In *In Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P 2017). To appear.*

[103] Accusations fly between uber and lyft. `http://bits.blogs.nytimes.com/2014/08/12/accusations-fly-between-uber-and-lyft/`.

[104] T. Vaidya, Y. Zhang, M. Sherr, and C. Shields. Cocaine noodles: exploiting the gap between human and machine speech recognition. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[105] T. Vu, A. Baid, S. Gao, M. Gruteser, R. Howard, J. Lindqvist, P. Spasojevic, and J. Walling. Distinguishing users with capacitive touch communication. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12, pages 197–208, New York, NY, USA, 2012. ACM.

[106] K. Wei, A. J. Smith, Y.-F. Chen, and B. Vo. Whopay: A scalable and anonymous payment system for peer-to-peer environments. In *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*, pages 13–13. IEEE, 2006.

[107] R. West. The psychology of security. *Commun. ACM*, 51(4):34–40, Apr. 2008.

[108] N. Xia, H. H. Song, Y. Liao, M. Iliofotou, A. Nucci, Z.-L. Zhang, and A. Kuzmanovic. Mosaic: quantifying privacy leakage in mobile networks. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 279–290. ACM, 2013.

[109] Xprivacy - the ultimate, yet easy to use, privacy manager for android. `https://github.com/M66B/XPrivacy#xprivacy`.

[110] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st USENIX conference on Security symposium*, pages 27–27. USENIX Association, 2012.

[111] Z. Xu, K. Bai, and S. Zhu. Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 113–124. ACM, 2012.

[112] H. Ye, S. Cheng, L. Zhang, and F. Jiang. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing &#38; Multimedia*, MoMM '13, pages 68:68–68:74, New York, NY, USA, 2013. ACM.

[113] K. Yoshida. Phonetic implementation of korean denasalization and its variation related to prosody. *IULC Working Papers*, 8(1), 2008.

[114] H. Zang and J. Bolot. Anonymization of location data does not work: a large-scale measurement study. In *Proceedings of MobiCom '11*, pages 145–156, New York, NY, USA, 2011. ACM.