Notice the Imposter! A Study on User Tag Spoofing Attack in Mobile Apps

Shuai Li, Zhemin Yang, Guangliang Yang, Hange Zhang, Nan Hua, Yurui Huang, Min Yang
Fudan University, China
{lis19@, yangzhemin@, yanggl@, 20210240152@, huan19@, 21210240204@, m_yang@}fudan.edu.cn

Abstract

Recent years have witnessed the rapid development of mobile services, spanning almost every field. To characterize users and provide personalized and targeted services, user tag sharing, which labels users and shares their data, is becoming increasingly popular. Its security attracts more and more attention, and a series of privacy issues have been reported in several specific services. However, up to now, there still lacked a thorough and comprehensive understanding of the characteristics and security of user tag sharing.

In this work, we conduct a systematic study of user tag sharing and its security. We first model user tag sharing with three phases, and discover that the privacy security issue commonly exists in practice. We generalize and formalize the privacy issue as user tag spoofing. Then, we propose a novel network-level smart fuzzing approach, called `UTSFuzzer`, against user tag spoofing. The key idea behind `UTSFuzzer` is to explore a large number of valid user tag values as input to imitate user tag spoofing against real-world mobile services. By applying `UTSFuzzer` on a large scale of real-world popular apps, we verify the effectiveness of `UTSFuzzer` and unveil that 100 mobile apps (including 115 mobile services) are vulnerable to user tag spoofing. The accumulated installations of all affected apps (users) reach more than 413 million. Additionally, `UTSFuzzer` shows user tag spoofing can cause serious attack efforts, including economic loss and user activity monitoring.

1 Introduction

Nowadays, modern services bring significant benefits to people's daily life and work by providing plentiful and diverse functionalities. To improve user experiences with targeted and personalized services, user tags [58] have been commonly utilized and applied in modern mobile apps. User tags are collected and constructed by service providers to describe various attributes or metadata of user profiles. Thus, they can characterize users and help mobile services understand various user behaviors.

Upon user tags, *user tag sharing*, or *tag sharing*, is becoming increasingly popular. This means, a user's tag data may be shared with other users. Consider two common real-world cases of user tag sharing. The first example is a ride-hailing and sharing app. When a user (rider) makes a ride, a set of nearby users (drivers) are presented with sharing their user tags, such as driver name, photo, rating, vehicle model, real-time GPS location, and even phone number (for further communication). Another example is a family-locator app (10,000,000+ downloads), which is commonly used for the elderly and children. When two users are located in the same family group, their user tags are shared with each other, including location information and safety status (e.g., car crash alert). User tag sharing has been supported by many popular apps. With the rapid development of machine learning, user tag sharing is commonly used in several typical cluster-based services [18, 23, 39, 58], such as interested content and friend recommendation.

As user tag sharing may involve user private data, its security raises concerns. It has been recently reported that several user tag sharing services may suffer privacy leakage issues [26, 44, 64]. For instance, Hagen et al. [26] demonstrated that a contact discovery service could be deceived by remote attackers to steal private user data. Nevertheless, prior work mainly focused on concrete cases. There is still a lack of a thorough understanding of the characteristics and security of user tag sharing. Several important research questions remain to be answered: How and what user tags are usually shared in practice? What is the root cause of the privacy issue? What are the attack surface and exploit vectors? How many real-world apps are vulnerable? What attack efforts may be introduced?

Motivated by these, we conduct the first comprehensive and systematic study on the characteristics and security of user tag sharing in modern mobile services. We first study the architecture and implementation of real-world tag sharing services and model the tag sharing process with three phases, including tag constructing, clustering, and sharing. Then, we analyze the security of tag sharing and find the privacy issue commonly exists in various tag sharing services. Based on this, we generalize and formalize this privacy issue as the *user tag spoofing* vulnerability, along with its root cause analysis and also the conclusion of three attack vectors. Our study demonstrates an attacker can easily exploit a user tag spoofing vulnerability by forging a fake tag and fooling the target tag sharing service. As a result, the victim service is deceived and wrongly shares the tags of the matched users. Specifically, when there is a user, one of whose tags is equal to the counterfeit tag, all tags of the victim user may be shared out by the victim service.

After understanding user tag spoofing attack, we further aim to learn its security impacts on real-world popular apps. For this purpose, a vulnerability detection tool against user tag spoofing is demanded. However, this is not an easy task, and it is hard to directly apply or extend existing techniques to achieve the goal. For static analysis based techniques, they face difficulties to understand user tag semantics in a concrete tag sharing implementation, as much tag information is dynamically set up, which requires interactions with the service side. For dynamic analysis based techniques, they can monitor and analyze the tag data shared with the service. Nevertheless, it is challenging for them to generate a valid forged tag capable to trigger the tag spoofing attack.

In this paper, we propose a novel fuzzing based security-vetting approach, called `UTSFuzzer`, that can infer the tag semantics of the target tag sharing service, and guide tag-mutation based fuzzing. Our `UTSFuzzer` approach is three-fold. `UTSFuzzer` first pre-processes given mobile apps with program analysis to filter the ones that do not enable user tag sharing. Based on our understanding of the sensitivity of tag semantics, the target tag employed for managing tag sharing is further located. Second, `UTSFuzzer` generates fake but valid test cases with an effective tag space exploration and tag mutation scheme. Third, by feeding these fake tags, `UTSFuzzer` vets the security of the target service by following the essence of user tag spoofing.

By applying `UTSFuzzer` on a large number of popular Android apps (from Google Play), we evaluate `UTSFuzzer`'s effectiveness and assess the security impacts of user tag spoofing. As a result, we find `UTSFuzzer` is effective with high precision (89.04% for filtering irrelevant apps and 95.00% for determining vulnerable mobile services), and 100 vulnerable apps along with 115 vulnerable user tag sharing services are identified. Note that there may be several concrete mobile services within one app. For instance, a shopping app may have a product recommendation service, payment service, and customer service. Furthermore, a large scale of users are affected which can be reflected by the accumulated installs of all affected apps (more than 413 million). Meanwhile, the severe consequences brought by user tag spoofing are demonstrated, including leakage of business secrets, the break of randomized preservation mechanisms, economic loss, and even monitoring of user activities. To mitigate the risks brought forward by user tag spoofing, we have responsibly informed the affected service providers. Many of them have confirmed our reports and timely fixed the reported vulnerabilities till the submission of this paper.

In sum, our contributions are outlined as follows:

- The security of user tag sharing mobile service and the generalized user tag spoofing attack against it are systematically analyzed, which complement the deficiency of the community's understanding in this regard.
- A novel fuzzing based security-vetting approach - `UTSFuzzer` is designed and implemented, which is scalable and precise to verify whether a mobile service is vulnerable to user tag spoofing or not.
- With `UTSFuzzer`, the landscape and severity of user tag spoofing attack in real world apps are unveiled. We responsibly informed the affected app developers and help them to timely fix the identified vulnerabilities.

2 Problem Statement

2.1 User Tag Sharing

To understand the characteristics of user tag sharing, we comprehensively study the design and implementation of a set of real-world user tag sharing services. As a result, we conclude the user tag sharing process with three key phases: tag constructing, tag matching, and tag sharing. Below we present more details about each phase and the related concepts (bound-tag and free-tag).

Tag constructing. For each user, a mobile app can collect various user information as user tags, and construct user fingerprints or portraits. User tags [58] are usually collected from plentiful perspectives, including user device data (e.g., phone number, location, device ID), registration and account information (e.g., identity and living address), and user app activities (e.g., interested content and joined groups). Thus, a user tag actually points to one of attributes or metadata of a user's profile. Normally, a use tag is a pair of key-value, e.g., <phone number, xxx-xxx-xxxx>. It is worth noting that user tag data can be dynamic and updated in real-time, as some user tags can be changed over time, e.g., GPS location and dwell time on different app content.

Tag clustering. With the constructed user tags, mobile services can better understand users' characteristics and perform user clustering. When several users have the same (or similar) user tag (denoted as *bound tag*), they are usually linked and managed together. For example, as introduced in §1, users in a family locator app will be clustered according to their family group ID tag.

Tag sharing. When some users are clustered with the same or similar bound tag, extra privileges may be assigned to them and allow them to (share) see each other's user tags. During the process of this tag sharing, it should be noted that many user tags apart from the bound tag (denoted as *free tags*) are shared. Taking the family locator app as an illustration, free

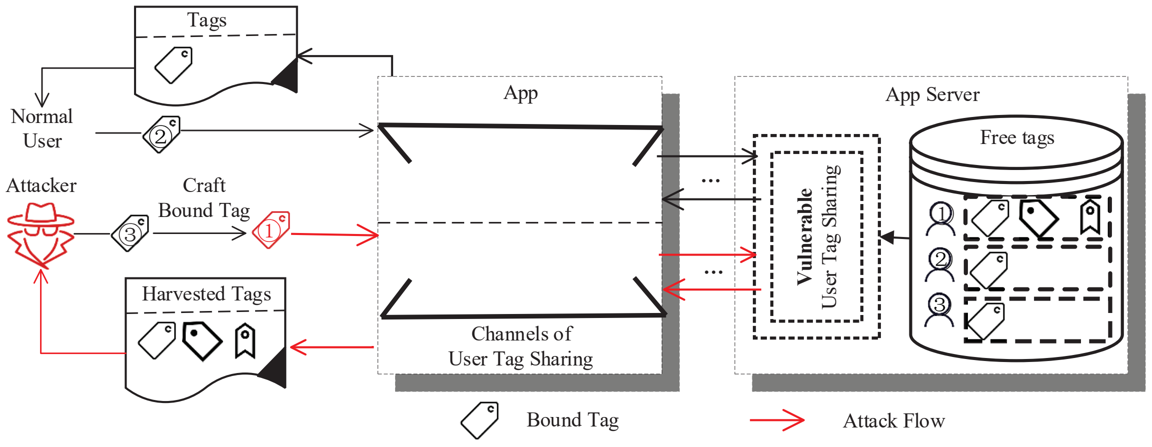


Figure 1: Attack flow of user tag spoofing.

tags such as live location and safety status are shared with users having the same bound tag (i.e., family group).

As discussed above, there are usually two key types of user tag, i.e., '*bound tag*' and '*free tag*', which play important roles in the three phases of user tag sharing. On the one hand, **bound tags** are selected from the constructed user tags, and act as the basis for clustering users. According to the specific functionalities a user tag sharing service provides, the service provider decides what user tags should be taken as bound tags. On the other hand, **free tags** point to all user tags but apart from the bound tags. Hence, user tag sharing is a process *when two users' bound tags are grouped together, their free tags may be shared with each other by the mobile service*.

To ease the understanding of user tag sharing and related concepts, a real example is Tinder, a high-profile dating app in Google Play store. First, Tinder collects various user data (e.g., geographical location, selfies, lifestyle, interests, and job information) for constructing user tags (Phase#1 Tag constructing). Then, its user tag sharing service - 'users discovery' relies on the geographical location (i.e., bound tag) to cluster users that are close to each other (Phase#2 Tag clustering). The bound tag here (i.e., geographical location) is picked according to its service logic, i.e., discovering nearby users. Finally, Tinder shares out each user's free tags with other users, e.g., selfies, interests, and lifestyle (Phase#3 Tag sharing).

2.2 User Tag Spoofing Attack

User tag sharing enhances and enriches mobile services. However, it raises security concerns. To understand the security of user tag sharing, we employ reverse engineering and conduct a manual analysis on a set of popular mobile apps supporting user tag sharing services in the real world. Our study discovers user tag sharing can be extensively abused by attackers for stealing user data and we generalize the security issue as user tag spoofing attack. The user tag spoofing vulnerability exists in different classes of modern mobile services. Its process is illustrated in Figure 1.

Particularly, an attacker can be a malicious user, who is familiar with how the target (victim) app works, and understands how and what user tags are used in the victim app. The attacker mainly targets user tag sharing services that cluster and manage users with a similar or same bound tag. For security reasons, the service providers enable data-masking and desensitization on the shared free tags, for example, by removing sensitive free tags or masking some tag values (e.g., replacing "alice@gmail.com" with "a****@g****.com"). However, many free tags are app-specific and may be shared with no protections (i.e., in plain text), such as a user's interested content and joined groups. The service providers need to make a good balance between user tag sharing and privacy protection. Nevertheless, this is a quite challenging task. Our study shows such a balance is rarely achieved in practice.

Upon the user tag sharing services, the goal of the attacker is to abuse the user tag sharing channels available in the victim app. To this end, the attacker creates a number of fake bound tags by subtly forging the values of their user data. Using these fake bound tags, the attacker can deceive the target service to share free tags belonging to other users. Our study further concludes three main attack vectors and discovers their security consequences. More details are presented below.

Attack Vector #1. This attack vector is straightforward. By tampering with the bound tag value, an attacker may be treated as another user and directly spoof the target user tag sharing services to steal the target user's private data. For example, when an attacker Mallory forges his own identity tag to be the one of Alice, he may successfully trick the vulnerable user tag sharing service into taking him as Alice, and thus illegally accesses the free tags that should only be shared with Alice.

Attack Vector #2. For tag clustering and matching, our study discovers that app developers usually manage and control the shared free tags in a multi-level permission system, e.g., group permission. For example, when Alice and Bob are grouped, they have the group permission to check partial and even all tags of all members in the group. When not grouped, tag sharing is not allowed. Therefore, in this attack vector, an attacker Mallory abuses the target service to be grouped with Alice and Bob, and obtains the group permission to access group members' free tags. More than that, different from conventional data security issues [10, 11, 33, 48, 59], which mainly focused on system-level data (e.g., device ID and phone number), the tag spoofing covers much app-level specific data. The sensitivity of this class of data is hardly determined. Hence, it is challenging for service providers to balance data sharing and protection.

Attack Vector #3. Many user tags seem insensitive and unimportant. These tags may not be well protected and shared out. However, by exploiting user tag spoofing vulnerabilities, an attacker may completely explore the target victim service, and obtain a large number of free tags. Benefiting from the shift from quantitative to qualitative changes, the attacker may obtain statistical data he is interested in. For instance, consider the leaked tag is the last active date (specific to apps that record when users are active), if the attacker can access all users' active date tags, the attacker may infer the number of daily/weekly/monthly active users of the target service, which are key business secrets for seeking funding and may be abused by competitors to build targeted market strategies.

Root Cause Analysis. As discussed above, one important reason is the improper protection of the shared app-specific user tags. App developers may ignore or not be able to figure out the potential privacy risks caused by user tag sharing. Furthermore, as the third attack vector discovers, some user tags seem to be unimportant but may have statistical characteristics, which greatly complicates the daunting problem of the proper protection of user tag sharing.

Moreover, different from traditional authentication issues that mainly focus on user identity, user tag spoofing targets authentication issues upon user tags. According to its attack vectors, the root cause of user tag spoofing attack lies in the missing or ineffective check of the authenticity of the provided bound tags. During the process of user tag sharing, the values of bound tags are collected from the client side, which can be controlled by the attacker and thus may be fake. Therefore, when receiving a request, the authenticity of bound tags should always be validated. However, as demonstrated in several vulnerable user tag sharing services [26, 44, 64], this is not an easy task since service providers need to balance between the user experience (or even service utility) and data security. Thus, the user tag spoofing issues commonly exist in modern user tag sharing services.

2.3 Real-World Example

The mobile app G[1] is a high-profile family locator app that has 10,000,000+ installs in Google Play store. Through app G, users can invite their family members and share their location information in a private circle only their family members can see. To join a private circle, a user needs to submit a valid circle code, which can be obtained by an online invitation provided by the circle owner. Typically, G would form specific user tags for a user, including username, battery status, last activity time, and most important - what circles he has. When the owner of a private circle decides to invite other family members to join, the relevant mobile service directly employs his `circle-id` to share back corresponding `circle-code` for joining his circle. Thereafter, the `circle-code` is provided to his family members to let them join.

G applies group-level permission to conduct access control and privacy protection. However, G is vulnerable to user tag spoofing. An attacker can exploit this vulnerable service by ① first transforming himself to be the owner of the victim circle by substituting the value of his `circle-id` tag (bound-tag) to be the id of the victim circle, ② then requesting the `circle-code` tag (free-tag) of the victim circle, and ③ finally joining in the victim circle with the illegally obtained `circle-code`. As a result of such user tag spoofing, the attacker can easily access the `circle-code` tag of a victim user and further leak other sensitive free tags about members of the victim circle, including their email addresses, device models, phone numbers, live locations and so on. With the leaked live location, the attacker can track the moving trajectory of victims, which can bring serious safety threats. Besides, by forging as many as possible `circle-id` tags, the scale of affected victims can be enormous.

[1]Note that we may be requested to anonymize the concrete (package) names of identified apps to avoid unnecessary influences.

2.4 Attack Formalization

The mobile service with user tag sharing is denoted as a class of functionality $F()$, which takes the bound tag of a user A_{bound_tag} as inputs and outputs free tags of other users U_{tags} according to A_{bound_tag}. Thus we get:

$$U_{tags} = F(A_{bound\_tag}) \quad (1)$$

Then, when the adversary conducts user tag spoofing attack, he would forge fake bound tag values - $A_{bound\_tag'}$, which is used in F to cluster users and control user tag sharing. Particularly, regarding an adopted type of bound tag, the adversary can refer to any available resource to construct a set S of probe values (i.e., counterfeit values), where the adversary picks one $A_{bound\_tag'}$ from it each time to conduct user tag spoofing attack. Thus, the attacker can get:

$$U'_{tags} = F(A_{bound\_tag'}), A_{bound\_tag'} \in S \quad (2)$$

With U'_{tags}, the user tag spoofing attack succeeds when the following condition satisfies:

$$\exists A_{bound\_tag'} \in S, U'_{tags} \neq NULL \,\&\, U'_{tags} \neq U_{tags} \quad (3)$$

2.5 Threat Model

Specifically, this paper considers the following as the threat model of user tag spoofing attack.

- **Attack target.** User tag spoofing takes the user tag sharing service providers as the attack target. Instead of attacking victims' devices or apps, the adversary seeks vulnerable user tag sharing services in the wild by conducting user tag spoofing on his own device.
- **Adversary**. Regarding user tag spoofing attack, we consider an adversary as a malicious requester who pretends to act as a normal user and is curious about the user tags of other users. By continuously forging the values of target bound tags and submitting them, the adversary harvests victims' data that he is not supposed to access.
- **Adversary goal**. Generally, the adversary is only allowed to access various free tags of users that share a similar or same bound tag with him. The attack objective is to deceive user tag sharing services and obtain as much as possible the detailed information of other users.
- **Adversarial capabilities**. We assume that the adversary is skilled in analyzing the target app and has complete control of his own mobile device. Namely, in practice, an attacker can first sign into the target app using his own account and pretend as a benign user. Then, he can monitor the app's mobile traffic with a rooted mobile device along with network proxy tools (e.g., MitmProxy [3]) and identify a bound tag. Next, he can mutate the bound tag with a fake value and craft a malicious HTTP request to send. Finally, the attacker can check whether the free tags of other users are shared out. The adversary is also assumed to be skilled in crawling user tags in scale, e.g., the ability to use script programming to conduct user tag spoofing attack in batch.

3 `UTSFuzzer`

3.1 Overview

After understanding the user tag spoofing security issues, we aim to comprehensively study the security impacts of user tag spoofing on real-world popular apps. To this end, we intend to build an automatic and precise vulnerability detection approach. The following goals should be achieved.

First, we need to identify the test target, i.e., candidate tag sharing services, from numerous mobile services in the wild. However, this is not an easy task. On the one hand, many user tag sharing services are highly bound with app functionalities and have diverse implementations. Manually exploring a large number of mobile services in wild required heavy manual effort. For automatic testing, the user tag semantics of a mobile service needs to be thoroughly understood.

Second, in line with user tag spoofing, the following critical point is creating fake but valid bound tags, which is necessary for spoofing the target services. Nevertheless, it is hard to properly deal with the exploration of the tag value space (introducing invalid values), and extensively handle different cases. First, for a bound tag, it can have different value formats in different mobile services, e.g., 'female' and '1' for gender. Second, it is hard to guarantee the forged bound tags (e.g., group id) fall in a valid value interval.

Third, we need to precisely determine whether a candidate mobile service is vulnerable or not. Particularly, by imitating user tag spoofing, the constructed fake bound tags are fed into the candidate mobile service, and the relevant service responses are obtained. In this condition, an effective solution for confirming whether the vulnerability exists is needed.

Motivated by the above goals, we propose a novel fuzzing based security-vetting system, called `UTSFuzzer`, that can infer the user tag semantics of the target services and guide tag mutation based fuzzing. Fuzzing is one of the most practical and popular security analysis techniques, and suitable to achieve our goals. The insight behind our `UTSFuzzer` fuzzing approach is that the processes of tag clustering and sharing are characterized by network packet content (i.e., corresponding HTTP/HTTPS requests and responses). Thus, to easily capture the used bound tags and shared free tags within user tag sharing services, `UTSFuzzer` is designed upon the network traffic of mobile services.

As shown in Figure 2, `UTSFuzzer` is a three-fold approach. First, `UTSFuzzer` preprocesses mobile apps with program analysis and filters apps that do not enable user tag sharing. For the remaining mobile apps, `UTSFuzzer` further locates the candidate user tag sharing services with an effective heuristic. Second, `UTSFuzzer` designs and applies a proper tag mutation strategy, which effectively explores the tag value space and generates fake but valid bound tags. Last, according to the formalization of user tag spoofing (Equation 3), a vulnerability determination module is designed to judge whether the tested candidate service is vulnerable or not. More details about the approach of `UTSFuzzer` are discussed below.

3.2 Preprocessing

3.2.1 Candidate Mobile App Detection

In this step, `UTSFuzzer` utilizes a key insight to filter out mobile apps that do not enable tag sharing services. It is noticed that user tag sharing services have specific code patterns. Based on the analysis of user tag sharing, the code pattern typically has two aspects. On the one hand, to ease the storage and usage of free tags that are shared by user tag sharing services, mobile apps typically manage them in one or several objects (e.g., an object of a Java class named `UserProfile`). On the other hand, to manage user sharing, a couple of bound tags are often sent out through the network APIs. With such a code pattern or practice of user tag sharing services, mobile apps that do not follow it can be discarded.

Following this code pattern, `UTSFuzzer` utilizes the bi-directional taint analysis to identify user tag sharing. In Particular, bi-directional taint analysis employs the forward taint analysis to track whether there exist user tags related objects

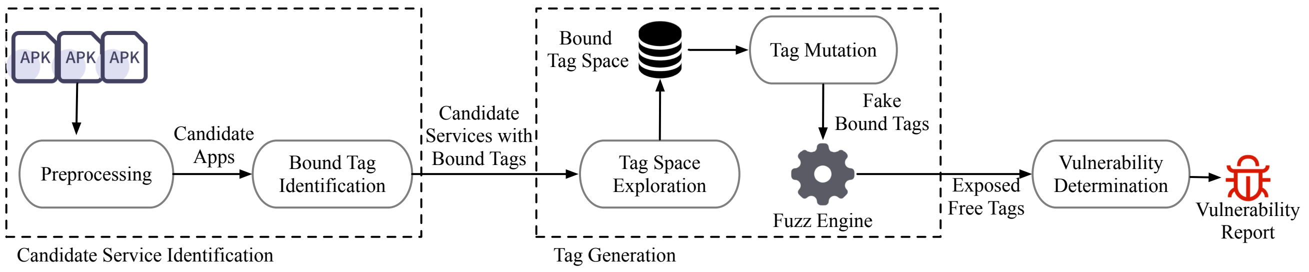


Figure 2: Overview of `UTSFuzzer`.

that are populated by network traffic. If none of any such object is found (i.e., no user tags are shared back), the analyzed app will be dropped. Meanwhile, through the backward taint analysis, the bi-directional taint analysis checks if any data is passed to the network request interfaces, whose responses are found to populate user tags related objects. If no data is identified to be sent out (then no bound tag is collected to the app server), the tested app is considered irrelevant with user tag sharing services and thus is dropped.

In the implementation, the bi-directional taint analysis mainly focuses on the semantics related to the widely-used network interfaces [6]. The bi-directional taint analysis takes the top six (as ranked by AppBrain [6]) network interfaces as taint sources, carefully supports their different usages according to their official documentations, and performs a classic data flow analysis to check whether an analyzed mobile app has the code pattern of user tag sharing. During the backward taint analysis, only data passed to form the URL paths or bodies of network requests is considered. Thus, parameters that are commonly seen and used to build headers of network requests are ignored (e.g., "Content-Type"). In the actual testing, `UTSFuzzer` would first exclude irrelevant apps that do not require `android.permission.INTERNET` permission (thus not being able to provide user tag sharing) and further filter out apps that are discarded by the bi-directional taint analysis.

3.2.2 Bound Tag Identification

Given a candidate mobile app, the next step of `UTSFuzzer` is to check if there are bound tags used in user tag sharing services, which are the potential fuzzing targets. As introduced in §2.1, bound tags play an important role in user tag sharing services. In user tag sharing services, the HTTP/HTTPS requests that contain bound tags are also our main fuzzing entry points. To this end, we start the dynamic testing on network traffic, which also have significant characteristics.

In this step, `UTSFuzzer` applies network packet analysis and identifies what is the employed bound tag for controlling the shared free tags in them. To do so, `UTSFuzzer` needs to understand the various semantics of bound tags and accordingly identify them. Our main idea is to take advantage of checking the sensitivity of bound tags. Many user tags are app specific. But differently, we find bound tags often have sensitive semantics, i.e., containing sensitive user data (usually a sensitive key). In other words, to make the provided services personalized or targeted, the services need to rely on bound tags that can precisely describe the characteristics of users, which are typically sensitive regarding semantics. Therefore, the sensitive semantic is a reliable sign of bound tags' existence. If one or several bound tags of a mobile service are identified, this mobile service along with the located bound tags is passed to the next phase as a candidate service.

Corresponding to this idea, `UTSFuzzer` effectively identifies the bound tags by iterating all parameters of the network requests that belong to the mobile services of candidate apps and subsequently checking the sensitivity of their semantics. During implementation, `UTSFuzzer` employs the app explorer 'DroidBot' [35] to trigger as many as possible mobile services of the candidate apps. Based on MitmProxy [3], `UTSFuzzer` captures the HTTP/HTTPS requests and responses of all triggered mobile services. Specifically, DroidBot is configured with the same exploration strategy (i.e., `BFS_Greedy` which is a customization of Breadth First Search) to exercise each candidate app for 30 minutes. This timeout is set according to the experiment settings as discussed in related works [27, 35, 53]. Besides, to effectively explore candidate mobile apps, we manually signed up and signed in all of them. Otherwise, most candidate apps would be stuck at registration or login during exploration, and numerous user tag sharing services may be missed.

3.3 Tag Generation Strategy

Given the candidate service with identified bound tags, `UTSFuzzer` applies tag mutation based fuzzing to vet its security. `UTSFuzzer` can generate fake but valid values for the bound tags, which will be taken as the inputs of fuzzing the candidate service. Generally, the tag generation strategy has two steps, including tag space exploration and tag mutation.

3.3.1 Tag Space Exploration

As the prerequisite, tag generation requires ensuring the validity of the generated bound tags. Otherwise, the candidate mobile service (even being truly vulnerable) typically would refuse the invalidly-crafted bound tag, causing the vulnerabilities to be missed. As discussed in §3.1, achieving this goal is not trivial, because the generated bound tag works only when its format and value are both valid.

To mitigate this problem, `UTSFuzzer` utilizes the key observations: 1) Many bound tags are shared as free tags in mobile services; 2) Tags values of existing users in the fuzzed service's app are supposed to be valid. Based on these key observations, a reliable source for exploring the value space of user tags comes from the network traffic of all mobile services within the fuzzed app. This means, for a mobile app pending testing, if any value of the identified bound tag is returned (i.e., the tag sharing process) through its services, these values should belong to its existing users and thus are supposed to be valid. Besides, we can also refer to user tag values that are mined in this way but from other apps. Since some types of bound tags have universal value formats (e.g., the email address should be like local-part@domain) and different apps may share some users, the values of user tags extracted from services of other mobile apps can be taken as a try-best scheme to explore the value space of user tags.

Based on these, a two-fold exploration scheme of user tag values is applied, which aims to mine as many as possible valid values for the identified bound tags. Particularly, to explore the value space of user tags as much as possible, the tag space exploration scheme contains two manners, i.e., in-app mining and cross-app mining. When a mobile app is being explored, the in-app mining seeks the values of bound tags from the network traffic of all triggered mobile services. By contrast, cross-app mining points to a process where bound tag values are mined from the triggered services of other apps (instead of the currently being tested mobile app).

When mobile apps of the candidate services are explored and as many as possible mobile services are triggered, `UTSFuzzer` monitors the data delivered back and checks if there are user tag values. As network responses of mobile services have a tree-like hierarchical structure (e.g., in the format of JSON or XML), if a type of user tag is identified at the leaf node, then all data of its sibling nodes is stored. This is due to the observation that the user tags shared by mobile services are typically well managed (instead of being scattered everywhere) in network responses to ease usage. By continuously doing so to all candidate apps that are selected by the bi-directional taint analysis, `UTSFuzzer` finally constructs a user tag value database stored in an instance of MongoDB [40] database, which represents the explored tag value space and will subsequently behave as a reliable source for bound tag generation.

3.3.2 Tag Mutation

After constructing the user tag value database with the two-fold tag space exploration scheme, a tag mutation strategy is proposed to complete the generation of bound tags. The details are as follows.

Particularly, the tag mutation mutates the original value of the identified bound tags from two aspects. First, if the user tag value database has multiple values for the identified bound tags, values extracted by in-app mining will be used in priority. Then if no needed user tag values mined from in-app mining are available, values explored by cross-app mining will be taken. The reason for designing such tag mutation is that user tag values mined by in-app mining must be valid and thus should be picked first. In contrast, user tag values mined from cross-app mining share an equal possibility to succeed since there is no prior knowledge. According to this aspect, when multiple values are available for mutating the identified bound tag, a configurable number of values will be randomly selected from them. Setting this configuration is mainly due to the consideration of keeping the balance between the chance of finding vulnerabilities and not affecting the quality of the tested mobile services.

Second, if the user tag value database has no values we need, tag mutation adopts a general mutation method to transform the original values of the identified bound tags. Namely, for numeric bound tag values, `UTSFuzzer` will mutate them to be adjacent ones (e.g., from "0" to "1"). For string or character type of bound tag values, `UTSFuzzer` will perform similar mutations according to alphabetical order. With these two aspects of mutating bound tag value, tag mutation will effectively generate forged bound tags and feed them into the tested candidate service.

It is worth noting that user tag values stored in the database are marked with the package names of the apps which they are mined from. Thus, `UTSFuzzer` can easily tell whether a user tag value is mined by in-app mining or cross-app mining. When it is the general way of mutating numeric and string type bound tags, tag transformation based fuzzing judges their value types by the type determination and conversion functions (e.g., the built-in method `isnumeric()` in Python).

3.4 Vulnerability Determination

When it comes to the determination of whether a tested mobile service is vulnerable to user tag spoofing attack, the confirming process is in line with the equation 3. During performing tag transformation based fuzzing, when a bound tag value A_{bound_tag} is mutated to be $A_{bound\_tag'}$ and the delivered free tags of other users U_{tags} changes to be U'_{tags} accordingly (i.e., $U'_{tags} \neq NULL$ & U'$_{tags}$ $\neq$U$_{tags}$), `UTSFuzzer` can determine that the fuzzed mobile service is vulnerable and subsequently generate a risk report. If the shared tags U_{tags} do not accordingly change with a counterfeit bound tag, tag transformation based fuzzing takes the next fake bound tag as input, till all of the configured numbers of fake bound tags are all tried or the vulnerability is confirmed.

In runtime, the test devices are all equipped with certificates of MitmProxy in their system certificate area. Thus, their network traffic can be monitored and intercepted. We also contact the authors of XPOChecker [34] and modified their tool to check the semantic sensitivity of bound tags. With the results of tag transformation based fuzzing, we program Python scripts to support the parsing of network responses in formats of JSON, XML, MessagePack [41] and Protobuf [24]. When a vulnerable service is identified, `UTSFuzzer` will automatically build the risk report, which includes the package name of the vulnerable app, the network traffic of the vulnerable mobile service, the adopted bound tag, the value of fake bound tags that trigger the vulnerability and the leaked free tags of other users.

3.5 Implementation

We implement a prototype system of `UTSFuzzer` for evaluation experiments. For the taint analysis module, it consists of 5,262 lines of Java code based on Soot [31] and FlowDroid [8] and 297 lines of Python code (for parsing logs). We build our fuzzing engine module from scratch. The engine can effectively monitor, replay and manipulate network traffic while automatically interacting with apps. This is implemented with 2,248 lines of Python code based on DroidBot and MitmProxy [3]. Our fuzzing engine is deployed on six test devices (OnePlus 9, Android 11) as our test bed.

4 Evaluation

In this section, we aim to understand the security landscape of user tag spoofing in real-world apps by answering the following research questions:

- RQ1: Is `UTSFuzzer` effective in terms of security detection?
- RQ2: How many real-world services are impacted by user tag spoofing?
- RQ3: What attack efforts may be introduced by user tag spoofing?

4.1 Experiment Setup

DataSet. Our empirical study is performed on a large dataset of apps collected from Google Play in April 2022. These apps were selected with the top 500 apps in each category listed by AppBrain [7] and AndroidRank [5] (30 categories were considered in total[2]). We crawled 25,901 unique app ids in total and successfully downloaded 25,158 apps while the remaining 743 apps failed to download due to area restrictions, payment requirements, and so on.

Hardware Environment. All experiments are conducted on a Ubuntu 18.04 LTS 64-bit server with 40 CPU cores (1.2GHz) and 128GB memory.

[2]Categories "Weather" and "Video Players & Editors" are filtered out since their apps are normally irrelevant with user tag sharing.

4.2 RQ1: `UTSFuzzer` Effectiveness

We apply `UTSFuzzer` on our dataset. Below we introduce the detailed analysis results. In the dataset, 23,505 apps are analyzed by bi-directional taint analysis, while the other 1,653 apps are dropped due to timeout, having no network permission, or failing to be analyzed by Soot and Flowdroid. As a result, 3,257 apps may contain candidate tag sharing service. Then, these apps are further analyzed by tag mutation based fuzzing. During the mining of user tag values, 1,428 apps were successfully tested and the constructed user tag value database has 76,486 keys and 740,040 values in total. The remaining apps failed to test due to the lack of necessary credentials for registration, servers not responding, compatibility issues, area restrictions, and strong SSL pinning.

Upon the kept apps, tag transformation based fuzzing was performed, and the configurable number of randomly selecting user tag values is set to 4 to balance the probability of successfully conducting user tag spoofing and the impact on the tested service. Finally, `UTSFuzzer` identified 100 unique apps (containing 115 mobile services) vulnerable to user tag spoofing with 11 different types of bound tags. Among the vulnerable services, the bound tag values that hit the vulnerabilities are separately mined by in-app mining (70), cross-app mining (5), and general mutation (40).

The bi-directional taint analysis of `UTSFuzzer` is performed in parallel and has a timeout of 30 minutes to analyze each app. This static process cost 1118 hours in total. The tag transformation based fuzzing of `UTSFuzzer` is performed with six Android phones (OnePlus 9, Android 11). It sets a timeout of 30 minutes for bound tag identification and tag space exploration. Another 30 minutes is set as the timeout of tag mutation. In total, the tag transformation based fuzzing of `UTSFuzzer` cost 1128 hours.

To ensure the reliability of our empirical study, it is critical to understand the performance of `UTSFuzzer`. Therefore, we evaluate each component of `UTSFuzzer`. Due to the missing ground truth, we need to manually verify the reported vulnerability results.

Bi-directional Taint Analysis. From all successfully analyzed mobile apps, the bi-directional taint analysis identified 3,257 candidate apps, whose category distribution is presented in Table 2. To validate its effectiveness, we sampled 15 apps in each app category separately from those identified as candidates and those not. Excluding the apps that were failed to explore (e.g., due to services being shut down, area restrictions, compatibility issues, and so on), we manually verified all the remaining 888 sampled apps. As shown in Table 1, the accuracy of bi-directional taint analysis achieves 89.04% true positive rate and 73.56% true negative rate. On the one hand, the main reasons for false positives are that the shared user tags located by forward taint analysis may not belong to real users (e.g., the phone number of real estate agencies) and there may exist redundant but actually ineffective code which complies with the principle of bi-directional taint analysis. On the other hand, the false negatives have many causes which include code obfuscation, adoption of customized WebViews, app packing, native encapsulation of service logic, unsupported network interfaces, and so on. Fixing these issues would be orthogonal to our study and since the overall effectiveness is practical, we consider referring to relevant state-of-the-art technologies to handle them respectively which we leave as future work.

Table 1: **Accuracy of `UTSFuzzer`.**

	#Num	TP	FP	TN	FN
Bi-directional	438	390	48	-	-
Taint Analysis	450	-	-	331	119
Tag Transformation	100	95	5	-	-
based Fuzzing	100	-	-	99	1

Tag Transformation based Fuzzing. To verify the efficacy of proposed tag transformation based fuzzing, we picked all 100 apps that are identified to be vulnerable and randomly sampled 100 ones that are not. By manually exercising them and imitating user tag spoofing attack, we check whether the reported apps along with their services are false positives and the not reported ones are false negatives. The verification results show that the true positive rate of tag transformation based fuzzing comes to 95.00% and the true negative rate comes to 99.00%. For the manually verified false positives, we found that all of them share user tags that actually belong to public institutions or public figures. For the one false negative, the reason is that DroidBot failed to trigger the vulnerable mobile service due to its limited exploration ability.

Since all the sampled apps for effectiveness validation are randomly and evenly picked, it is firmly believed the verification results can well provide a reasonable effectiveness estimation of `UTSFuzzer` among the whole dataset. Besides, as `UTSFuzzer` achieves high precision and recall, we consider it reliable to perform our empirical study.

4.3 RQ2: Vulnerability Detection

Impacted mobile apps & users. In total, `UTSFuzzer` identified 100 apps along with 115 mobile services that were vulnerable to user tag spoofing attack, and the affected app category distribution is shown in Table 4. The most affected app categories are "Dating" (56) and "Social" (20), which are consistent with the common sense that mobile services of apps within these two categories normally involve user tag sharing. Additionally, the scale of affected mobile users is estimated by accumulating the installs of all affected mobile apps, which reach an astonishing number - 413M+. Although the installs of mobile apps do not represent the real number of their users, the sum of their installs actually can reflect how big the affected user scale is.

Impacted tag sharing services. While belonging to user tag sharing in essence, these 115 affected mobile services have different semantics, which cover many common service scenarios. As shown in Table 3, there are 11 types of user tags in total that were employed in these vulnerable mobile services. Although normal users cannot tamper with their user tags, providers of these vulnerable services ignore that the attacker can manipulate his user tag by intercepting service traffics. In contrast to them, some secure implementations of user tag sharing are also found during the performance validation of `UTSFuzzer`. These secure implementations of user tag sharing ensure the authenticity of employed user tag by storing user tags on the server and directly obtaining their values from the server side (instead of from the client side), which is an effective way that can inspire the developers of affected mobile services to get rid of the vulnerabilities.

Impacted user tags. During our empirical study, we find that various user tags can be leaked due to user tag spoofing attack. As shown in Table 3, the leaked user tags include demographic information (e.g., age and gender), geographical location, device information, contact information, education information, employment information (e.g., job and income), health information (e.g., height and weight) and so on. Besides, the leaked user tags can be app-specific. For instance, the leaked `parent-id` is specific in family locating services, which should be considered sensitive since the attacker can take advantage of it to infer the parent-child relationship among users. Furthermore, it should be pointed out that many leaked user tags are actually overshared by vulnerable services ((such oversharing is discussed as XPO risks in [34])), which strengthens our mining scheme of user tag values and may also bring about serious privacy risks.

iOS Security. To confirm whether mobile apps on the iOS platform are affected, we manually checked the counterparts of 100 vulnerable apps identified during our empirical study. Finally, 46 apps were confirmed to have iOS versions and 36 of them were able to be tested on an iPhone 11 device. The failure reasons are mainly due to network error, being checked as an unsafe environment, crashes, and so on. For the successfully tested iOS apps, we manually confirmed that 35 apps of them were affected and shared the same vulnerable services with their Android counterparts. The left one app cannot be verified as its vulnerable mobile service provided on the Android platform cannot be triggered on the iOS platform. The results highly suggest that user tag spoofing attack is independent of mobile platforms (i.e., both Android and iOS users are similarly affected.) and the scale of affected users can be larger when considering all mobile platforms.

4.4 RQ3: Attack Efforts and Case Study

Generally, conducting user tag spoofing leads to the illegal access of user data and even the leakage of user tag database of a mobile app when being conducted in scale. Moreover, it

Table 2: The detected number of mobile apps in each app category that have candidate user tag sharing services.

Category	**#Num**	**Category**	**#Num**	**Category**	**#Num**	**Category**	**#Num**	**Category**	**#Num**
Art & Design	39	Auto & Vehicles	99	Beauty	42	Books & Reference	76	Business	135
Comics	43	Communication	82	Dating	176	Education	90	Entertainment	140
Events	80	Finance	135	Food & Drink	200	Health & Fitness	143	House & Home	116
Libraries & Demo	15	Lifestyle	147	Medical	120	Music & Audio	76	News & Magazines	206
Parenting	93	Personalization	22	Photography	40	Productivity	84	Shopping	209
Social	174	Sports	181	Tools	63	Transportation	33	Travel & Local	198

Table 3: Vulnerability details for randomly picked apps with each type of employed user tag. Note that a type of user tag may have various names while they actually are the same, e.g., account_id, user_id, member_id, person_id and profile_id all refer to the identifier tag of users. Thus, only one unified name for a type of user tag is presented.

Bound Tag		**Package**	**#Installs**	**Service Description**	**Tag Generation Strategy**	**Samples of Leaked Free Tags**
id	user_id	c**.e***	10M+	Get users' homepage	In-app	job, income, children, education, ethnicity, smoking, alcohol, height
	room_id	c**.m***.t***	500K+	Get owners of chat rooms	General	age, gender, country, language, income
	language_id	c**.f***.c***	1M+	Get users via language	In-app	distance, birthday, date of creation, is_online
	author_id	c**.m***.d***	1M+	Get authors of artworks	In-app	biography, artworks, museums, date of death & birth
	circle_id	c**.g***.f***	10M+	Get users in a circle	General	deeplink, email address, parent_id, device model, phone number
	moment_id	a**.t***.d***	500K+	Get commentators	General	birthday, country, city, email address, phone number
email address		c**.t**.v***	500K+	Get users' homepage	In-app	country, region, birthday, gender, date of creation
country		c**.w***.b***	100K+	Get live streaming users	General	name, country_id, rate, video_id
phone number		c*.h***.m***	10M+	Get users of contacts	Cross-app	real first name, real last name, date of last activity & registration
date		j*.c*.a***.a***	500K+	Get current popular users	General	age, country, login_date, height, weight, distance
location		r*.t***.a**	1M+	Get nearby users	Cross-app	car, birthday, zodiac, region, height, latitude & longitude, date of last activity & creation

Table 4: The category distribution of affected apps.

Category	#Num	Category	#Num
Art & Design	1	Auto & Vehicles	1
Books & Reference	2	Business	1
Comics	1	Communication	2
Dating	56	Education	6
Entertainment	2	Food & Drink	1
Health & Fitness	1	House & Home	1
Lifestyle	6	Medical	1
Parenting	3	Shopping	2
Social	20	Sports	2

is noticed that user tag spoofing can bring about more severe results than direct privacy leakage.

4.4.1 Business Secret Leakage

App P is a popular platform sharing local news and videos for users in India, which has 10,000,000+ installs in Google Play store. As identified to be vulnerable by `UTSFuzzer`, its service checking profiles of members is affected by user tag spoofing. Specifically, the employed user tag (i.e., "`member_id`") has numeric values, which enable the adversary to easily forge probe user tag values. As shown in Figure 3 (a), the attacker can simply iterate the value of `member_id` to access user tags of all other users in app P. More importantly, it is identified that the sensitive user tags of employees of app P are leaked when the `member_id` is forged to be "0" in value, including the phone number, email address, and geographical location

```
GET /pvrest-2/restapi/
member/0?xxx HTTP/1.1
…
```

```
HTTP/2 200 OK
…
{
   "id":               "0",
   "name":             "***",
   "organization":     "P",
   "mobileNumber":     "***",
   "longitude":        "***",
   "latitude":         "***",
   "deviceId":         "***",
   "dob":              "***",
   "gender":           "M",
   "imageUrl":         "***",
   "email":            "***",
   "verified":         "true",
    …
}
```

(a)

```
GET /api/profile/6374d4bb9afc/
xxx HTTP/2
…
```

```
HTTP/2 200 OK
…

{
   "name":             "***",
   "followers":        [***],
   "following":        [***],
   "devices":          [***],
   "currentDevice":    "***"
   "createdAt":        "***",
   "phone":            "***",
   "deviceId":         "***",
   "allInstalledApps":
                 [com.***, …],
   "isOnline":         false,
    …
}
```

(b)

Figure 3: (a) Privacy leakage of employee users in app P; (b) Compromised mechanism of randomized user tag in app W.

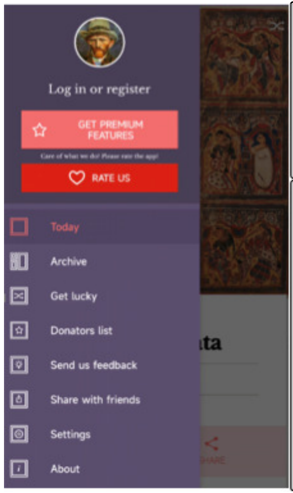
of the employee. Such private data should be considered a business secret since it can enable the adversary to conduct social engineering attack, which is a common penetration route to infiltrate the internal network of modern companies and organizations [45, 55].

More than that, we found many vulnerable mobile services also leaked the date of registration and last active date of users. Therefore, the attacker can conduct user tag spoofing against these vulnerable mobile services in scale to leak the registration time and the last date of being active of all other users. Such leakage enables the attacker to understand the number of online users in real-time, daily registered users, daily active users (DAU), and so on of a mobile service, which can be key business secrets for seeking funding and even be analyzed by competitors to build targeted marketing strategies.

4.4.2 Breaking Preservation Mechanisms

In contrast to the above situation, there exist mobile services that employ randomized user tag (thus the value of the randomized user tag can not be guessed in theory), which is a preservation mechanism to protect from being attacked. App W is a gaming social platform that has installations of more than 10 million in Google Play. Compared to app P, it employed a randomly generated user tag (i.e., profile-id) to control the sharing of other user tags, which seems to be more secure. However, as mined by UTSFuzzer, valid values of such randomly generated user tag are found to be shared in other triggered service traffics of app W. Specifically, the attacker can take advantage of "nearby users" and "following/-follower" services to continuously harvest profile ids of all users in app W. As shown in Figure 3 (b), the adversary can thereafter conduct user tag spoofing and crawl numerous user tags of other users including phone numbers, devices, and installed apps of victims. Thus, such a preservation mechanism of randomized user tag actually fails to achieve its goal.

4.4.3 Causing Economic Loss



```
POST /APIMobile/artworks/
query HTTP/1.1
…
{
   "local_date":"2022-11-17",
   …
}
```

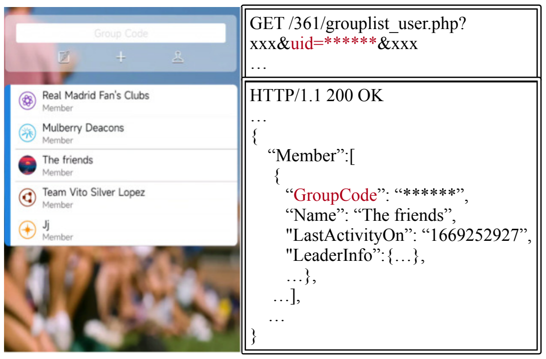
```
HTTP/2 200 OK
…
{
   "entities":[
     {
       photo_related_info,
       author_info,
       museums_info
     },
     …
     ]
   …
}
```

Figure 4: The vulnerable "Today" service in app D.

App D is a popular community with 1,000,000+ installs in Google Play store, which connects art lovers with classic, modern, and contemporary art masterpieces. As the premium subscription of app D regulates - "there are over 750 artists in our database. Want to see all of them? Unlock all features", users can check artworks in this app but only a limited number of artists (and so as the artworks, museums) are provided for unsubscribed users. However, as shown in Figure 4, UTSFuzzer found that the attacker can simply forge his user tag - local_date to deceive the "Today" mobile service which is designed to share users with limited artworks according to their current date. By exploiting this vulnerability, the adversary can easily bypass the premium subscription mechanism of app D and access data that he is not supposed to obtain. Moreover, the attacker is even able to sell these data to users of app D to make a profit, which can bring unacceptable economic loss to app D.

4.4.4 Monitoring User Activities

App T is a popular platform with 500,000+ installs in Google Play store, which benefits users to manage and instantly communicate with their team, club, or group. A user in app T can create his own group or join other users' groups with relevant group codes. As designed by app T, a user can check the messages, event schedules, shared photos or files, and members of groups that he is a member of. Typically, a user can only access information of users that joined the same groups as him. When a user signs in app T, there is a user tag sharing mobile service that relies on his user tag to automatically share back information about his groups on the main page of



```
GET /361/grouplist_user.php?
xxx&uid=******&xxx
...
```

```
HTTP/1.1 200 OK
...
{
  "Member":[
   {
     "GroupCode": "******",
     "Name": "The friends",
     "LastActivityOn": "1669252927",
     "LeaderInfo":{...},
     ...},
   ...],
  ...
}
```

Figure 5: A vulnerable user tag sharing service in app T.

app T. Specifically, as shown in Figure 5, this mobile service relies on `uid` (i.e., the employed user tag) to fulfill its functionality. However, by forging counterfeit `uid`, the attacker can transform himself to be any user with a specific `uid` tag and illegally obtain information about the victim's groups, including the group code, date of last activity, leader information, and so on. Furthermore, with the leaked group codes, the attacker can join the groups of the victim without permission. Therefore, by conducting user tag spoofing in scale, the attacker is even able to monitor the members, events, chat messages, shared photos, or files (if there exist) in any group. Based on these, the adversary can easily monitor user activities and even bring physical threats to victims (e.g., assaulting users who participate in an event the attacker objects to).

4.5 Ethics & Responsible Disclosure

Conducting the empirical study might raise ethical issues. Therefore, we carefully manage our research activities to ensure that they stay within legal and ethical boundaries. In this work, we followed the established practices as discussed in [2, 64]. In particular, `UTSFuzzer` was configured to test at the speed of how a normal user acts, and each mobile service was fuzzed at most four times. This helpfully avoids impacting existing service quality (e.g., service availability). Meanwhile, `UTSFuzzer` was set to automatically check the HTTP status code of service response and stop immediately once the exceptional status is detected, e.g., Internal Server Error (HTTP code 500). In our experiments, such exceptional status was never detected. Moreover, user tags involved in our empirical study were managed at the granularity of a single user tag for preventing them from being linked (to identify real users) and stored inside an internal storage server with password control. While performing case studies, we created an attacker account and a victim account that both belong to us for verifying whether the tested mobile service is vulnerable or not. When accessing real user data was inevitable, the relevant case study was performed after being authorized.

We have conducted responsible disclosure of our findings to the app developers and actively worked together with them to fix these problems. To ease the overhead of handling our reports, we briefly explained user tag spoofing and directly pointed out the vulnerable service along with the employed user tag. Note that the responsible disclosure is private (only available to relevantly affected app developers). The vulnerable mobile apps along with sensitive user tags involved in this paper are all anonymized or masked.

5 Limitation & Discussion

The main research focus of academia has been illegal privacy collection for a long time. In contrast, this paper takes mobile apps as the entry point of studying user tag sharing services and systematically analyzes the user tag spoofing attack against them. By adopting the user tag mining scheme with hints provided by mobile apps, it is shown that numerous mobile users are at risk of user tag spoofing. It should be noted that `UTSFuzzer` can be extended to test upon a broader software against unauthenticated web APIs.

Admittedly, our approach naturally suffers from certain limitations, which will bring both false positives and false negatives. Regarding the false positives, as inherited from the limitations of static program analysis, bi-directional taint analysis of `UTSFuzzer` can bring false positives when locating candidate user tag sharing services. Besides, the modified XPOChecker [34] may also occasionally result in false positives when it treats non-user data (e.g., the geographical location of a supermarket) as user tags.

Additionally, `UTSFuzzer` can have false negatives. First, the limited exploration ability of existing dynamic exercisers can result in the insufficiency of exploring user tag value space and the missing of vulnerable mobile services. Second, the strong SSL pinning [32] was seen in a few tested apps, which can prevent MitmProxy from monitoring and intercepting their network traffic and thus bring false negatives. Third, mobile apps that did not request `"android.permission.INTERNET"` permission were directly dropped since it is assumed that user tag sharing services should be able to access the internet to communicate with their servers. But this may not hold in several rare situations, e.g., apps that simply use another 'server' app residing on the same device through inter-app communication. Besides, `UTSFuzzer` may also miss bound tags that have no known or apparent sensitive semantics and bring false negatives. Nevertheless, it should be noted that since bound tags are used to cluster users, they are typically unique enough to represent a group of users (i.e., distinguish from other users) and thus are sensitive in most cases. Moreover, it should be noted that there are some inevitable factors bringing false negatives. For example, some mobile apps need special credentials (to register or log in), devices (e.g., real IoT devices needed to connect with the companion app), networks of a specific area, or other requirements to be effectively explored.

Despite being affected by these limitations, `UTSFuzzer` achieves practical performance. It should be noted that UTSFuzzer can directly benefit from any improvement in these aspects and since handling these limitations is orthogonal to this study, we leave them as our future work.

Furthermore, we would like to discuss the difficulty of user tag exploration in practice and the mitigation against user tag spoofing. The difficulty of tag exploration for spoofing lies in the size of tag value spaces. For example, for the gender tag, the attacker can mutate its value in a relatively narrow space. But for the UUID tag, it is normally hard to generate effective values. To mitigate this problem, UTSFuzzer adopts several strategies (e.g.,in-app mining) to maximize the success rate. Regarding the mitigation strategies, we summarize three ones based on our study results and the developers' feedback: 1) Authenticity verification against HTTP requests should be enforced to check whether a bound tag's value belongs to the service requester; 2) Aapp developers should reduce tag involvement in their services, lessening attack surface; 3) Risk control should be introduced against abnormal behaviors.

6 Related Work

Fuzzing. Fuzzing is an effective security vetting technique. Generation-based fuzzers [56, 62] normally generated inputs, for example, based on input templates. Nevertheless, without domain knowledge specific to each mobile app, it is hard to obtain such input templates available for generating valid user tag values. Guided fuzzers [1, 16, 36, 47, 52] applied heuristics to improve their capability of triggering unexpected program behaviors, e.g., mutating the seeds that help explore new program branches. But, these heuristics cannot be applied to our research problems. The reason is for a user tag value that successfully triggers spoofing, its mutated ones are not necessarily able to achieve the same goal. Mutation-based fuzzers [13, 15] explored programs with continuous input mutation, e.g., random mutation (e.g., flipping and truncating) of inputs. They expected to hunt unexpected program behaviors. Nonetheless, they hardly hold in the context of this paper. The reason is that the simple mutation of a user tag value does not necessarily generate another valid user tag value. For example, a valid email address - "alice@gmail.com" in an app is truncated to "a@gmail.com", which may not be valid in this app. Therefore, existing fuzzing approaches were not designed for verifying user tag sharing security and are hardly extended to solve the research problems in our paper. In contrast, UTSFuzzer relies on the proposed mining scheme to effectively explore the valid value space of user tags and further verify the existence of user tag spoofing risk. Furthermore, due to ethical considerations, UTSFuzzer is designed to be non-exhaustive and thus cannot be fairly compared in performance with traditional fuzzers.

Mobile Privacy Security. There have been numerous researches focusing on the security of mobile privacy for more than a decade. A significant amount of efforts have been put into the identification of mobile privacy [9, 28, 29, 42, 46], detection of privacy collection with static taint analysis [8, 14, 25], dynamic taint analysis [20, 38, 54] and network traffic analysis [19, 49], or even the legality of privacy collection against user intention [17, 21, 22, 43, 60, 61] and privacy policy [4, 51, 57, 65]. Unlike these works, this paper pays attention to the security issues of user tag sharing instead of privacy collection related risks.

Mobile Service Authentication. Prior work in this area focused on the authentication problem of mobile services, such as single-sign-on (e.g., [50]), password protection ([37]), authentication key management (e.g., [66]), authentication design (e.g., [12]) and so on. Different from them, UTSFuzzer focuses on the vulnerability of user tag sharing mobile services and we demonstrate that existing security protection solutions were limited against user tag spoofing attacks.

Management of Mobile User Data Sharing. Actually, there are two main research directions in this aspect. The first class of work focused on judging whether the user data shared by mobile services is appropriate. Koch et al. [30] and li et al. [34] revealed that if the shared user data is more than the presented ones (i.e., user tag oversharing) in mobile apps, serious privacy leakage can be brought. Compare to them, we systematically studied the user tag spoofing issue rather than discussing whether user tags are overshared. The second direction focused on the research target of our paper, i.e. the security of user tag sharing. A few prior works have revealed a series of tag security issues, e.g., ride-hailing service [64], nearby users service [44, 63] and contact discovery service [26]. Another related work was conducted by AuthScope [67], which performed a study on post-authentication security in mobile services. Different from prior work, our paper provides a thorough and comprehensive understanding of user tag spoofing attack and reveals its landscape along with its severe security consequences.

7 Conclusion

In this paper, we conduct a systematic and comprehensive study on the security of user tag sharing. We generalize and formalize the user tag spoofing security issues, along with the root cause analysis and the conclusion of their attack vectors. We propose a novel tag mutation based fuzzing approach, called UTSFuzzer, that can identify tag sharing services and vet their security. With a large-scale empirical study of 23,505 real-world apps, we reveal the severity of user tag spoofing attack, which can bring various serious consequences and put numerous mobile users at serious risk. We believe our work can facilitate the understanding of user tag sharing, improve its security and inspire future research.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Key Research and Development Program (2021YFB3101200), National Natural Science Foundation of China (62172104, 62172105, 61972099, 61902374, 62102093, 62102091), Natural Science Foundation of Shanghai (19ZR1404800), China Postdoctoral Science Foundation (BX2021079, 2021M690706). Zhemin Yang was supported in part by the Funding of Ministry of Industry and Information Technology of the People's Republic of China under Grant TC220H079. Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics Systems, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

References

[1] American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.

[2] Vulnerability Disclosure Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Vulnerability_Disclosure_Cheat_Sheet.html.

[3] Cortesi Aldo, Hils Maximilian, and Raumfresser. Mitmproxy - an interactive HTTPS proxy. https://mitmproxy.org/.

[4] Benjamin Andow, Samin Yaseer Mahmud, Justin Whitaker, William Enck, Bradley Reaves, Kapil Singh, and Serge Egelman. Actions speak louder than words: Entity-sensitive privacy policy and data flow analysis with policheck. In *Usenix Security Symposium (USENIX Security)*, 2020.

[5] AndroidRank. Free Android Market Data, History, Rankings. https://www.androidrank.org/.

[6] AppBrain. Android network libraries. https://www.appbrain.com/stats/libraries/tag/network/android-network-libraries.

[7] AppBrain. Google Play Ranking: The Top Free Overall in the United States. https://www.appbrain.com/stats/google-play-rankings/.

[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 2014.

[9] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228, 2012.

[10] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. On demystifying the android application framework:{Re-Visiting} android permission specification analysis. In *25th USENIX security symposium (USENIX security 16)*, pages 1101–1118, 2016.

[11] David Barrera, H Güneş Kayacik, Paul C Van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 73–84, 2010.

[12] Antonio Bianchi, Eric Gustafson, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. Exploitation and mitigation of authentication schemes based on device-public information. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 16–27, 2017.

[13] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.

[14] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2015.

[15] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *2015 IEEE Symposium on Security and Privacy*, pages 725–741. IEEE, 2015.

[16] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[17] Xin Chen and Sencun Zhu. Droidjust: Automated functionality-aware privacy leakage analysis for android applications. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2015.

[18] Yanbo Chen, Jingsha He, Wei Wei, Nafei Zhu, and Cong Yu. A multi-model approach for user portrait. *Future Internet*, 13(6):147, 2021.

[19] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. Obfuscation-resilient privacy leak detection for mobile apps through differential analysis. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2017.

[20] William ENCK. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[21] Hao Fu, Zizhan Zheng, Somdutta Bose, Matt Bishop, and Prasant Mohapatra. Leaksemantic: Identifying abnormal sensitive network transmissions in mobile applications. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2017.

[22] Hao Fu, Zizhan Zheng, Aveek K Das, Parth H Pathak, Pengfei Hu, and Prasant Mohapatra. Flowintent: Detecting privacy leakage from user intention to network traffic mapping. In *International Conference on Sensing, Communication and Networking (SECON)*, 2016.

[23] Mengke Gao, Yan Zhang, and Yue Gao. Research progress of user portrait technology in medical field. In *Proceedings of the 2nd International Symposium on Artificial Intelligence for Medicine Sciences*, pages 500–504, 2021.

[24] Google. Protocol Buffers. https://developers.google.com/protocol-buffers?hl=zh-cn.

[25] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information flow analysis of android applications in droidsafe. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2015.

[26] Christoph Hagen, Christian Weinert, Christoph Sendner, Alexandra Dmitrienko, and Thomas Schneider. All the numbers are us: Large-scale abuse of contact discovery in mobile messengers. In *NDSS*, 2021.

[27] Yuyu He, Lei Zhang, Zhemin Yang, Yinzhi Cao, Keke Lian, Shuai Li, Wei Yang, Zhibo Zhang, Min Yang, Yuan Zhang, et al. Textexerciser: feedback-driven text input exercising for android applications. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1071–1087. IEEE, 2020.

[28] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. {SUPOR}: Precise and scalable sensitive user input detection for android apps. In *Usenix Security Symposium (USENIX Security)*, 2015.

[29] Jianjun Huang, Xiangyu Zhang, and Lin Tan. Detecting sensitive data disclosure via bi-directional text correlation analysis. In *International Symposium on Foundations of Software Engineering (FSE)*, 2016.

[30] William Koch, Abdelberi Chaabane, Manuel Egele, William Robertson, and Engin Kirda. Semi-automated discovery of server-based information oversharing vulnerabilities in android applications. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 147–157, 2017.

[31] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (Cetus)*, 2011.

[32] Indusface Learning. What is SSL Pinning? – A Quick Walk Through. https://www.indusface.com/learning/what-is-ssl-pinning-a-quick-walk-through/.

[33] Muyuan Li, Haojin Zhu, Zhaoyu Gao, Si Chen, Le Yu, Shangqian Hu, and Kui Ren. All your location are belong to us: Breaking mobile social networks for automated user location tracking. In *Proceedings of the 15th ACM international symposium on Mobile ad hoc networking and computing*, pages 43–52, 2014.

[34] Shuai Li, Zhemin Yang, Nan Hua, Peng Liu, Xiaohan Zhang, Guangliang Yang, and Min Yang. Collect responsibly but deliver arbitrarily? a study on cross-user privacy leakage in mobile apps. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1887–1900, 2022.

[35] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 23–26. IEEE, 2017.

[36] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 627–637, 2017.

[37] Siqi Ma, Juanru Li, Surya Nepal, Diethelm Ostry, David Lo, Sanjay Kumar Jha, Robert H Deng, and Elisa Bertino. Orchestration or automation: authentication flaw detection in android apps. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2165–2178, 2021.

[38] Björn Mathis, Vitalii Avdiienko, Ezekiel O Soremekun, Marcel Böhme, and Andreas Zeller. Detecting information flow by mutating input data. In *32nd International*

Conference on Automated Software Engineering (ASE), 2017.

[39] Ruomu Miao and Benqian Li. A user-portraits-based recommendation algorithm for traditional short video industry and security management of user privacy in social networks. *Technological Forecasting and Social Change*, 185:122103, 2022.

[40] MongoDB. MongoDB: The Developer Data Platform. https://www.mongodb.com/.

[41] Msgpack. MessagePack: It is like JSON. but fast and small.. https://msgpack.org/index.html.

[42] Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and XiaoFeng Wang. Uipicker: User-input privacy identification in mobile applications. In *Usenix Security Symposium (USENIX Security)*, 2015.

[43] Xiang Pan, Yinzhi Cao, Xuechao Du, Boyuan He, Gan Fang, Rui Shao, and Yan Chen. Flowcog: context-aware semantics extraction and analysis of information flow leaks in android apps. In *Usenix Security Symposium (USENIX Security)*, 2018.

[44] Iasonas Polakis, George Argyros, Theofilos Petsios, Suphannee Sivakorn, and Angelos D Keromytis. Where's wally? precise user discovery attacks in location proximity services. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 817–828, 2015.

[45] Nia Quinn. Riot Games gives social engineering attack update. https://esports.gg/news/league-of-legends/riot-games-announce-social-engineering-attack-update/, 2023.

[46] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2014.

[47] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[48] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 50 ways to leak your data: An exploration of apps' circumvention of the android permissions system. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 603–620, 2019.

[49] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. Recon: Revealing and controlling pii leaks in mobile network traffic. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2016.

[50] Shangcheng Shi, Xianbo Wang, and Wing Cheong Lau. Mossot: An automated blackbox tester for single sign-on vulnerabilities in mobile applications. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 269–282, 2019.

[51] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D Breaux, and Jianwei Niu. Toward a framework for detecting privacy policy violations in android application code. In *International Conference on Software Engineering (ICSE)*, 2016.

[52] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[53] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 245–256, 2017.

[54] Mingshen Sun, Tao Wei, and John CS Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[55] Tessian. 15 Examples of Real Social Engineering Attacks. https://www.tessian.com/blog/examples-of-social-engineering-attacks/.

[56] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594. IEEE, 2017.

[57] Xiaoyin Wang, Xue Qin, Mitra Bokaei Hosseini, Rocky Slavin, Travis D Breaux, and Jianwei Niu. Guileak: Tracing privacy policy claims on user input data for android applications. In *International Conference on Software Engineering (ICSE)*, 2018.

[58] Xu Wang, Xiong Wei, Jiajun Ma, Guangjie Li, and Jiancun Zuo. User portrait technology and its application scenario analysis. In *The 2021 3rd International Conference on Big Data Engineering*, pages 64–69, 2021.

[59] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. Android permissions remystified: A field

study on contextual integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 499–514, 2015.

[60] Shengqu Xi, Shao Yang, Xusheng Xiao, Yuan Yao, Yayuan Xiong, Fengyuan Xu, Haoyu Wang, Peng Gao, Zhuotao Liu, Feng Xu, et al. Deepintent: Deep icon-behavior learning for detecting intention-behavior discrepancy in mobile apps. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[61] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[62] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2139–2154, 2017.

[63] Fanghua Zhao, Linan Gao, Yang Zhang, Zeyu Wang, Bo Wang, and Shanqing Guo. You are where you app: An assessment on location privacy of social applications. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 236–247. IEEE, 2018.

[64] Qingchuan Zhao, Chaoshun Zuo, Giancarlo Pellegrino, and Li Zhiqiang. Geo-locating drivers: A study of sensitive data leakage in ride-hailing services. In *Annual Network and Distributed System Security symposium, February 2019 (NDSS 2019)*, 2019.

[65] Sebastian Zimmeck, Ziqi Wang, Lieyong Zou, Roger Iyengar, Bin Liu, Florian Schaub, Shomir Wilson, Norman Sadeh, Steven M Bellovin, and Joel Reidenberg. Automated analysis of privacy requirements for mobile apps. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2017.

[66] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. Why does your data leak? uncovering the data leakage in cloud from mobile apps. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1296–1310. IEEE, 2019.

[67] Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. Authscope: Towards automatic discovery of vulnerable authorizations in online services. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 799–813, 2017.