Perplexed Messengers from the Cloud: Automated Security Analysis of Push-Messaging Integrations

Yangyi Chen[1]*, Tongxin Li[2]*, XiaoFeng Wang[1], Kai Chen[1, 3] and Xinhui Han[2]

[1]Indiana University Bloomington
[2]Peking University
[3]Institute of Information Engineering, Chinese Academy of Sciences

ABSTRACT

In this paper, we report the *first* large-scale, systematic study on the security qualities of emerging push-messaging services, focusing on their app-side service integrations. We identified a set of security properties different push-messaging services (e.g., Google Cloud Messaging) need to have, and automatically verified them in different integrations using a new technique, called *Seminal*. Seminal is designed to extract semantic information from a service's sample code, and leverage the information to evaluate the security qualities of the service's SDKs and its integrations within different apps. Using this tool, we studied 30 leading services around the world, and scanned 35,173 apps. Our findings are astonishing: over 20% apps in Google Play and 50% apps in mainstream Chinese app markets are riddled with security-critical loopholes, putting a huge amount of sensitive user data at risk. Also, our research brought to light new types of security flaws *never known before*, which can be exploited to cause serious confusions among popular apps and services (e.g., Facebook, Skype, Yelp, Baidu Push). Taking advantage of such confusions, the adversary can post his content to the victim's apps in the name of trusted parties and intercept her private messages. The study highlights the serious challenges in securing push-messaging services and an urgent need for improving their security qualities.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification–*Validation*; D.2.5 [**Software Engineering**]: Testing and Debugging–*Code inspections and walk-throughs*

General Terms

Security

Keywords

mobile push-messaging services; Android security; mobile cloud security; security analysis

*The names of the first two authors are in alphabetical order.

CCS'15, October 12-16, 2015, Denver, CO, USA
 ISBN 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813652.

1. INTRODUCTION

Push-messaging (aka., *cloud-messaging*) has emerged as a major channel that connects software vendors to their mobile users, which is now provided by all commercial clouds (Google, Amazon, etc.), device manufacturers (Samsung, Apple, Microsoft, etc.), and increasingly third-party providers (e.g., Urban Airship [12], PushIO [9]). These services are further integrated by *syndicators* across different platforms, allowing the vendor to push a message to all devices associated with different *push clouds* such as Google Cloud Messaging (GCM), Amazon Device Messaging (ADM) and Apple Push Notification service (APNs) in one step. Through this channel, popular apps (Facebook, Skype, Yelp, Netflix, etc.) receive sensitive information (private messages, bank account balances, etc.) and commands for security-critical operations (e.g., erasing data on lost devices) from their app servers. Therefore, its security and privacy assurance is of critical importance.

Hazards in the cloud messenger. A recent study, however, casts serious doubts on the security qualities of the channel [28]: researchers preliminarily investigated GCM, ADM and two other services through *manual analysis*, and revealed that they were all riddled with security-critical loopholes. For example, a lapse in GCM allowed an unauthorized party to hijack a victim's *registration ID* (an identifer for the app on her device) and receive all the messages that should be pushed to the victim. On the device side, improper use of `PendingIntent` was found to cause this secret token to be exposed, allowing unauthorized parties to intercept the victim's messages and inject data to her app.

Even though all these flaws have since been fixed [28], the gripping concern remains whether they are just a tip of the iceberg. After all, over 40 providers and syndicators are already in this booming business and new players continue to join the party, whose services have been integrated into millions of apps. Less clear here is whether they also suffer from any existing or new threats never known before. Most importantly, nothing is out there today to help service providers and app vendors identify vulnerabilities in development and integration of push-messaging services, and third parties evaluate their security protection. As a result, the security qualities of these services are hard to assure.

Detecting vulnerable messaging integrations. As the first step toward better protecting push messaging, we developed in our research the first technique to help detect security-critical vulnerabilities in device-side integrations of these services. We focus on their integrations within apps, as these components are most likely to be the weakest link of this type of cloud services, as indicated by the prior research [28]. Also, never before have the push-messaging integrations been thoroughly investigated. More specifically we first identified a set of properties for a secure integration. Automatic

verification of these properties, however, turns out to be challenging: given an app, little is known about the location of its code segment related to a push-messaging service, its detailed functionalities (registrations, receiving messages, etc.), the way messages are passed from the service's SDK to the app (e.g., through callback, static or dynamic receiver, etc.), the models of the cloud service integrated (e.g., syndication, with or without a service app) and the further operations the app performs on a received message. To address this challenge, our new technique, called *Seminal* (secure messaging integration analysis), utilizes the simple sample code offered by each cloud-messaging service to automatically recover such semantic information and guide the analysis on both the SDK and the app's processing of messages (Section 3). Seminal is designed *specifically* for analyzing the push-messaging service, one of the most important mobile-cloud service in use. Our evaluation shows that the approach can effectively detect security weaknesses within popular apps' integrations in a large scale.

Our findings. Our implementation of Seminal was used in a study to analyze the SDKs of 30 push-messaging services, covering most third-party services we are aware of, and over 35,173 commercial apps (e.g., Facebook, Yelp, etc.). Our findings are astonishing: 57% of the SDKs (17 out of 30) have at least one security-critical flaw and 50.2% of the apps contain vulnerabilities that could be exploited by an unauthorized party to either inject messages or intercept sensitive user information. In addition to a better understanding about the scope and magnitude of known flaws, which has never been studied at this scale, many *new* security flaws were brought to light by our study, which are causing confusion within existing push-messaging services and often fundamental to their design. For example, we found that almost *all* the apps integrating multiple cloud-messaging services (e.g., GCM, ADM, etc.) do not discriminate which service a received message actually comes from: Facebook and Skype integrate both GCM and ADM services, while on most (if not all) mobile devices only one of these services is supported; therefore, a malicious app on the same device can leverage the Intent receiver for the unused service to inject messages into these apps, in the name of a party the app user trusts (e.g., her close friend) (Section 4.1). Another important observation is that all the cloud-messaging services can only identify the device and the app, *not* the user supposed to receive a message. As a result, over 50% of social, finance and health apps, including the popular ones such as Yelp and Pinterest, were found to have a user confusion problem, which potentially allows a malicious party to impersonate a friend of the victim and push messages to her through a different user account (Section 4.1).

In addition, our research reveals the pervasiveness of client identifier (CID) exposure, a serious threat since CID is an authentication token and its disclosure enables the adversary to directly contact a cloud server to obtain a user's messages. The problem was found in over 6.5% of the 17,557 apps sampled from 5 major Chinese app markets. Furthermore, a new power-saving strategy of the Baidu Push (an extremely popular Chinese push-messaging service) was shown to allow a malicious app to intercept the messages delivered to their recipients (Section 4.1). Altogether, we discovered 17,668 vulnerable apps and have reported the most critical issues to related parties. The importance of these new findings has been well acknowledged by the industry: e.g., Facebook rewards us $2000 for the security flaws our approach detected in their app. A demo of the attacks is posted on a private website [11]. We also report a measurement study on these vulnerable apps (Section 4.2).

Contributions. The contributions of the paper are summarized as follows:

• *New technique for detecting vulnerable integrations*. We came up with the first methodology for a systematic analysis on push-messaging integrations. A simple technique we built leverages sample code as a guidance to efficiently detect potential security problems within service SDKs and the apps integrating them. This new approach can be utilized to identify security-critical vulnerabilities when developing new apps and the client components of a new push-messaging service.

• *New findings*. Running our tool, we performed a comprehensive study on the security properties of most third-party SDKs and over 35,173 apps. This study reveals the gravity of the situation: many SDKs and a large portion of the apps contain security flaws *never known before*, with serious consequences once exploited. This points to an urgent need to improve the security qualities of push-messaging integrations.

2. BACKGROUND

Push-messaging services. To receive messages from a push cloud, an Android app is given a unique identifier, which we call *registration ID* for those associated with device manufacturers' clouds such as GCM and ADM, or *client ID* (*CID*) for the identifiers generated and used by other push-messaging services in a way similar to a *universally unique identifer* (*UUID*). The identifier is known to the application vendor's *app server* and the push cloud's *connection servers*. To push a message to a user, the app server asks the cloud to deliver it to the app with an identifier associated with the user. This *message delivery process* ultimately goes through a connection server that directly talks to the app or indirectly through an Android service app (e.g., `android.gms`). To obtain such an identifer, an app first needs to register with the service whenever a new user logs in. This *registration process* involves communication with the cloud, which either causes the connection server to generate a registration ID or sends a locally constructed CID (by the service SDK integrated into the app) to the server. The identifier is also submitted by the app to its app server (typically through an out-of-the-band SSL connection between them) to bind it with a user ID. Such an infrastructure and operations vary across different services, including those provided by device manufacturers or the third parties, and those integrating others' services (Figure 1), as explicated below:

• *Manufacturer push-messaging service*. Examples of the services for Android devices in this category include GCM, ADM, Samsung Push Service, Nokia Notification Service, etc. As illustrated in Figure 1, they are provided by manufacturers' clouds, whose connection servers directly talk to the application vendor's app server. A prominent feature of such services on the device side is a service app (e.g., Android service `android.gms`, the ADM client, etc.) that relays the message received from the connection server to the target app, using Android Inter-Process Communication (IPC). Also through the service app, an app registers with the push-messag ing service (sometimes through an SDK integrated within the app), which involves authenticating the device to the connection server and requesting the server to create a registration ID. Note that the ID here is bound to the app vendor's server: only this app server is allowed to push messages to the app.

• *Third-party services*. In addition to device manufacturers, third parties also start providing their own push-messaging services. For example, Baidu Push [2], Getui [6] and JPush [7] are already serving hundreds of millions of users. Such services are similar to their manufacturer counterparts, except that the connection server directly talks to the service SDK within the app through a socket connection. In this way, the SDK authenticates the app to the server

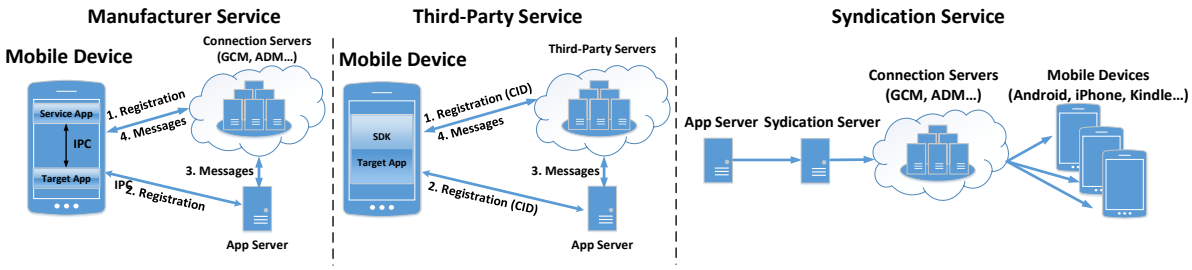

Figure 1: Push Messaging Services

using its CID (which different from registration ID, also serves as a secret token) before getting messages from the server.

• *Push-messaging syndication*. To support convenient message pushing across different devices and platforms (Nexus, Samsung, Kindle, etc.), *syndication services* (*syndicator* for short) like Urban Airship [12], Push Woosh [8], etc. have emerged in recent years. Such a service is framed over multiple manufacturer clouds, such as GCM, ADM, using a syndication server that connects to the app server the service provider sets up for each of those clouds. To use the service, the app vendor runs her app server to push a message, together with the recipient's CID, to the syndication server, which retrieves the registration ID associated with this CID for a specific manufacturer cloud, and then forwards the message to the syndicator's app server there. After that, the connection server deliveries it to the end user's app.

Service integrations. To get a push-messaging service, the app developer often needs to integrate its SDK to her app. As discussed above, the SDK functions as a messaging interface, interacting with a connection server to receive messages from the app vendor, and then handing them over to the app. It is also in charge of generating a CID for the app and sending it to the app server or the syndication server during a registration. The communication between the SDK and cloud servers goes through socket, using the CID as an authentication token in the case of third-party services or service syndications, while its service to its hosting app is often delivered through Android IPC or a callback function registered for the messaging event (Section 3.2). Also, for the app integrating a manufacturer push-messaging service, it also uses IPC to receive messages from the service app (e.g., `android.gms`).

The IPC communication on Android mainly relies on the *Intent* mechanism. An Intent is a message that describes the operations to be performed by its recipient. The operations here include invoking a user interface (`startActivity`) or a service (`startService`), or broadcasting to the *receivers* associated with a specific *action* as specified by recipient apps. Such a receiver can be *private*, so that only its hosting app is allowed to send messages to it, or *protected by permission* to ensure that only an authorized party (with a proper permission) can communicate with it. Otherwise, the receiver is made public. Such settings typically show up within an app's manifest file but can also be programmatically defined during the app's runtime. A problem for this Intent mechanism is that the sender app's identity is not given to the recipient. When it becomes necessary for an app to find out where a message comes from, a typical solution is using `PendingIntent`, a token that enables whoever receives it to perform a set of pre-defined operations with the sender's identity and permission. Examples of such operations include `startActivity`, `startService`, etc. To find out who sends an Intent, the recipient app can simply call `getTargetPackage` or `getCreatorPackage` with the token to get the sender app's package name from the operating system. Prior research on four push-messaging services (GCM, ADM, Urban Airship and a Chinese push cloud [28]) shows that `PendingIntent` is often broadcasted and therefore can be intercepted by a malicious app to invoke private functionalities within the sender app. However, the study does not go beyond this on the push client side. Little has been done to find out whether any other security weaknesses, particularly those related to the design of push messaging, exist on the mobile device, not to mention any effort to systematically detect them. This is the focus of our research.

Adversary model. We consider a malicious app running on the victim's device. The malware does not have a system privilege but requires a set of dangerous permissions, such as `READ_PHONE_STATE` and `INTERNET`, to exploit push-messaging services the victim's apps use. These permissions are extensively utilized by legitimate apps and therefore claiming them is not considered to be suspicious. Further, we assume that the adversary is capable of setting up an app server with the push clouds serving the victim, opening an account with her application vendor and also having a mobile device for the attack purpose.

3. SEMINAL

Security properties and challenges in automatic checks. The security goals of a push-messaging service can be described as follows: (1) a message for a user from an app server should only be pushed to that user and no one else; (2) a user only gets her own message (not others) from the authorized app server (not from an unauthorized party). To this end, a set of properties are expected from the client-side integration. Specifically, the app integrating the service should communicate with the service app (in the case of the manufacturer service) or the SDK (in the case of third-party or syndicated service) through an *authenticated secure channel*. Further it should check whether an incoming message belongs to the current user (according to the User ID in the message). Note that this verification can only happen within the app, as no other entities in the service can differentiate two users sharing the same app on the same device, which if not handled properly can lead to serious security breaches (Section 4.1). Finally, the CID (the authentication token for the third-party or syndicated service) should always be kept secret. These properties were identified from the way current push-messaging services work, as elaborated in Appendix.

Automatic verification of these properties within real-world apps is by no means trivial. Specifically, given an app, it is less clear what service has been integrated there, manufacturer-based, third party, syndication or sometimes even a combination of multiple services. Further, we need to identify the interface between the app and the service SDK within the app's APK automatically to understand whether an incoming message has been sent by the SDK and received by the app in a secure way. New techniques need to be developed to carve out a small set of integration-related code from an app for an automatic, efficient property checking. Following we

elaborate the design and implementation of Seminal that addresses these challenges.

3.1 Overview

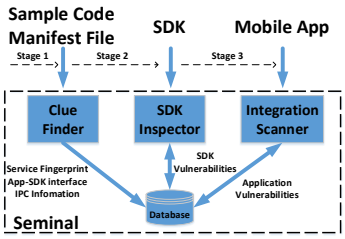

Figure 2: Seminal Design and Architecture

At a high level, Seminal performs a progressive three-stage analysis on the integration of a push-messaging service, with each stage handling increasingly complicated code and providing supports for the next stage, as illustrated in Figure 2. At the first stage, only a few hundred lines of sample code are inspected, together with its manifest file. The purpose here is to collect a set of information for the next-stage analysis on the service's SDK. Most important here is an interface for an app integrating the service to interact with the SDK (e.g., receiving a push message from the SDK). Such an interface is a program component within the app, called *anchor*, which could be a broadcast receiver, service or a callback function. Other information gathered at this stage is a fingerprint for uniquely identifying the service (e.g., an action name for a specific third-party service).

The follow-up step uses the anchor to inspect the way that messaging related communication is authenticated within the SDK and the protection of the CID. After all mainstream services go through these two stages, what have been found are kept in a database, with each service's information indexed by their fingerprints. Then, at the third stage, Seminal scans a large number of apps, automatically identifying the services they incorporate according to the fingerprints, and utilizing their anchors to guide the analysis of their interactions with the SDKs and the security checks that happen to User IDs within their integrations.

Architecture. The architecture of Seminal is described in Figure 2. Our design includes a *clue finder*, an *SDK inspector* and an *integration scanner*. The clue finder automatically examines the sample code and its manifest file, collecting information for the follow-up analysis. Using the information, the SDK inspector further checks the SDK for security properties (authentication and CID protection). The results are then stored in the database and used by the integration scanner to detect security weaknesses within apps.

Example. Figure 3 presents a simplified version of the sample code for PushIO [9], a popular syndication service. Given the code, the clue finder first inspects its manifest file. Under the definition of the receiver `PushIOBroadcastReceiver`, the standard GCM action `c2dm.intent.RECEIVE`, together with a different receiver specified in the code, indicates that the service is a syndicator over GCM, and the service name `PushIOGCMIntentService` is identified as a fingerprint for the service (Section 3.2). Then, through a quick analysis of the code, the clue finder discovers that the PushIO SDK talks to its hosting app through a dynamic receiver `mBroadcastReceiver`, which serves as an interface between the hosting app and the SDK (i.e., the anchor). Based upon such information, the SDK inspector checks the IPCs within the SDK, between `PushIOGCMIntentService` and that for interacting

Manifest File Snippets:

```
<receiver android:name="com.pushio.manager.PushIOBroadcastReceiver"
        android:permission="com.google.android.c2dm.permission.SEND">
        <intent-filter>
                <action android:name="com.google.android.c2dm.intent.RECEIVE" />
                <action android:name="com.google.android.c2dm.intent.REGISTRATION" />
                <category android:name="com.pushio.basic" />
        </intent-filter>
</receiver>
...
<service android:name="com.pushio.manager.PushIOGCMIntentService" />
```

Sample Code Snippets:

```
public class PushSettings extends Activity {
   private BroadcastReceiver mBroadcastReceiver;
   ...
   @Override
   public void onResume() {
      super.onResume();
      mBroadcastReceiver = new BroadcastReceiver() {
         @Override
         public void onReceive(Context context, Intent intent) {...}
         registerReceiver(mBroadcastReceiver,
                        new IntentFilter("com.pushio.basic.PUSHIOPUSH")); }} ...
}
```

Figure 3: Simplified Sample Code and Manifest File

Inputs	Manifest File, Sample Code, Manufacturer Actions
Service Type	*Syndication*: service (not in sample code)+manufacturer action *Third-party*: otherwise
Fingerprint	Service name that is defined in SDK (not in sample code)
Anchor	Check sample code for: `BroadcastReceiver`, `registerReceiver`, Serivce (`onHandleIntent`) or Callback function

Table 1: Clue Finder Logic

with the anchor, to detect any improper setting of the communication (Section 3.2). Also, since the service involves a CID, its generation and use are then evaluated to detect leaks. All the information collected here is further used to analyze an app integrating such a service, which is identified with the fingerprint. More specifically, from `onReceive` of the anchor, the scanner runs an integration-specific approach to find out whether User ID has been checked in the app (Section 3.3). It also inspects the app's code and manifest to ensure that all the receivers are configured properly. Any problem discovered during this process is reported as an integration flaw.

Implementation. We implemented Seminal over FlowDroid, a static analysis tool [14]. Our current implementation is largely automatic, capable of analyzing tens of thousands of commercial apps (Section 4.2). However, it still contains some manual steps, including collection of the inputs to the system (sample code, SDKs and apps), label of the CID within an SDK (Section 3.2) and validation of the results reported by Seminal (Section 3.2). How to further automate these steps is left for the future research.

3.2 Clue Finding and SDK Analysis

Clue finding. Seminal is designed to inspect the service's sample code, finding "clues" for its operations. Almost every push-messaging service provides a set of sample code (aka., *demo code*), typically a few hundred lines with a manifest file, to demonstrate the way the service should be used within an app (e.g., the example in Figure 3). The specific clues we are looking for include the service type and fingerprint, and the anchor of its integration within an app. Such semantic information can be recovered in a fully automatic way, as summarized in Table 1.

Specifically, the clue finder maintains a list of action names that uniquely characterize the small set of manufacturer services (GCM, ADM, Samsung and Nokia), e.g., `c2dm.intent.RECEIVE`. For the sample code from other service providers (third party or syndication), it is fingerprinted by the service name defined in its manifest files but does not show up in the sample code, e.g., `PushIOGCMIntentService` in Figure 3. Such a service component is in-

cluded in the SDK and therefore cannot be changed by the app developer. The type of the push-messaging service is also determined from the manifest: if it specifies a manufacturer's action and a service component (not in the code), then sample code is for a syndicator; otherwise, it belongs to a third-party service. This is because a manufacturer service, like GCM and ADM, does not define any service component in the manifest of its sample code, and instead directly runs a separate service app to deliver messages.

Finding anchors and the way an app gets messages is equally straightforward. An anchor gets messages from the SDK through one of the following channels: Intent broadcast, service or callback. To determine which channel has been used, our approach automatically generates an abstract syntax tree (AST) to search for a receiver object within the sample code. At most one such object is there for handling the message pushed from the SDK or the service app. It can be easily identified from the declaration of a class extending `BroadcastReceiver`, together with a specification of its `onReceive` method. Also, the presence of `registerReceiver` indicates that the receiver is dynamically generated during an app's runtime. If such elements (`BroadcastReceiver`, `onReceive`) are missing in the code, the clue finder then searches for the method for the service component (`onHandleIntent`) defined there. If they are also missing, the chance is that the push-messaging service is integrated through a user-defined callback function invoked by the SDK. The function is discovered when the name of a newly created object or a class name (`AtomPushNotifyCallback` in Figure 4) also becomes a parameter of a function (`setAtomPushNotifyCallback`), as illustrated in Figure 4. Once the IPC receiver, service or callback is found, it is automatically labeled as the anchor of the integration.

```
AtomPush.setAtomPushNotifyCallback(new AtomPushNotifyCallback() {
   @Override
    public void OnPushNotifyArrive(String messagetTitle, String messageContent, String
                                   subContent, HashMap<String, String> params) {...}});
```

Figure 4: Simplified User-defined Callback Function

A problem here is that the name of the anchor could be changed by the app developer. Identifying it across different apps relies on some invariants that characterize the anchor. For the IPC, the action it uses for Intent filtering is such an invariant, which is either a constant string (e.g., `cn.jpush.android.intent.MESSAGE_RECEIVED`) or constructed through concatenating the app's package name with a string, a convention for defining an action that gets inputs from the SDK. The constant string here serves to identify the action and then the anchor. In the absence of such an action, the class the anchor extends is used as the invariant. For a callback function, the name of the API for delivering the handler back to the SDK is used to find out the anchors employed by different app developers (e.g., `AtomPush.setAtomPushNotifyCallback` in the example). After such inspections are done, all the findings are saved to a record in a database, which can be located by the fingerprint of the service.

Authentication in SDK. Based upon the clues gathered, Seminal continues to check the security properties within the SDK, particularly the authentication on the IPC communication from where the message gets in (from the service app) to where it is handed over to the app (i.e., the anchor). Authentication here relies on Android's security settings: the receiver should be either private or protected by permission, and all the IPC calls should target a specific package. Although verification of such settings has also been done in the prior research [15], Seminal is designed to check the calls related to push messaging, according to how the SDK works.

Specifically, our SDK analyzer (implemented in our research using Soot [10]) looks at the anchor and then goes backward along the execution path through which a message is transmitted from the entry point of the SDK to its exit (to the app), checking the setting of each IPC (e.g., `startService`) and its handler (e.g. `onReceive`) one by one. The analyzer first finds the IPC call ipc within the SDK that sends an Intent to the anchor. The relation between the call and its handler is established using class or action name, whose constant part can be directly found within ipc or through a simple tracing of the define-use chain using Soot. Then, the analyzer locates ipc on the control-flow graph (CFG) of a handler, which corresponds to another IPC ipc'. This indicates that ipc' happens within the SDK before ipc is later made to the anchor. After that, the analyzer further goes through all the IPC calls on the SDK's AST to find out ipc', using the action of the handler or its class name, which also shows up in the calls like `startService`[1]. This backtracking process continues until the current handler is found to use the standard manufacturer action such as `c2dm.intent.RECEIVE`, indicating that it is the entry point for the SDK (that receiving messages from a system service app like `gms`), or the IPC call discovered from the AST cannot be located within any handler's CFG, which only happens in a third-party SDK, where messages directly come from the connection server through a socket connection.

All the IPCs found in this way are then inspected for their security settings, as discussed above. Particularly, when a receiver is found to be protected by a permission, we further find out which party is given the permission: it is supposed to be either a system app or the hosting app[2]. Also Seminal checks whether there is any attempt to broadcast `PendingIntent`, and if so, whether any part of its content is left blank (so the adversary can fill in his operation [28]). An alarm is raised when such a problem is found.

CID secrecy. As mentioned earlier, the CID is an authentication token for a third-party or syndication service and it secrecy is therefore of critical importance. Since the token is generated by the SDK and only used there, our SDK analyzer is also tasked to check whether it is well protected. Specifically, the function that creates the CID is always highlighted in the instructions for integrating the service as the first thing the app developer needs to know. What we did in our research is to label this function within the SDK and let our analyzer take care of the rest. The analyzer starts from the output of the function, which is the newly created CID, to perform a backward slicing, following the define-use chain generated by Soot, until this analysis ends at a set of APIs, like `getIMEI` and `Random`. Android does provide an API, `uuid`, for creating a secret token. The function returns an 128-bit string. However, as discovered in our research, many SDKs just uses a set of public information (e.g., `DeviceID`) collected by the APIs such as `getDeviceId`, to produce the CID (Section 4.1). In this case, the token is no longer a secret. Our analyzer just inspects all the APIs it observes: if all of them can be invoked by other apps (with a proper permission) to get the same resource and produce the same result, it concludes that the CID generation is not secure.

Further, the SDK analyzer goes through the AST to identify all the occurrences of the CID. From each occurrence, our approach performs a taint analysis using FlowDroid to find out whether the content of the CID shows up at any IPC call. Once found, the call

[1]Linking an IPC to its handler has been extensively studied [30]. Our approach is a simple version of these techniques, which is tailored to how push-messaging SDKs work.

[2]This is done manually in our current implementation.

is inspected: if its Intent is broadcasted to an action, our system reports that a leak has been found.

3.3 Integration-Specific App Checking

Based upon what it collected from sample code and SDKs, Seminal is ready for scanning apps for integration flaws. Here we elaborate its unique design for a high-performance analysis, which avoids going through the whole app, and just focuses on its integration part.

Receiver checking. The first thing that needs to be done on the app side is to make sure that its anchor has been securely configured. Here the anchor we are concerned about is a static or dynamic receiver, as the callback channel cannot be accessed by the adversary outside the app's process. Specifically, given an app whose push-messaging service is identified by its fingerprint, the integration scanner first locates its anchor based upon the action claimed by the receiver (recorded in the database). For the static receiver, an inspection of the app's manifest file reveals its protection level. As to the dynamic receiver, which cannot be configured as private, the scanner checks whether a permission is in place to authorize the sender, and whether it is only given to legitimate apps.

```
if (localVineSingleNotification.recipientUserId != this.mAppController.getActiveId())
    SLog.e("This message is intended for someone else {}.",
            Long.valueOf(localVineSingleNotification.recipientUserId));
```

Figure 5: User ID Verification Example

User ID filtering. To determine whether an app has properly verified an inbound message's user ID, the integration scanner performs a taint analysis, starting from the anchor (the taint source), to track how the content of the Intent (the message) is handled by the app. Such content processing is rather straightforward within an app, typically involving a format check, sometimes a classification (e.g., determining whether the message is an advertisement or a private one), then issue of a notification and post of the message's content. The security check on User ID is detected from such operations on "tainted" data (that is, the content propagated from the source), using a *behavior signature*. Specifically, we look at a set of comparisons of equality observed early in the processing of the message, between a tainted string or long integer (that is, the user-related information from the message) and a variable holding nonconstant value stored in the app (the ID of the current user), as illustrated by the example in Figure 5. The idea here is based on the observation that such a comparison is distinctive during the processing of an incoming message, because in the case of the format checking and classification, equality comparisons always happen between some message fields and a set of *constants* such as action and message types (e.g., `com.google.android.c2dm.intent.RECEIVE`). In the example, `receipientUserId` is a string from the taint source (the Intent) and is found to be compared with the output of `getActiveId`. Through a simple define-use analysis, the scanner finds that the latter returns the content of a variable stored in an object. This indicates that a user-ID checking happens. Indeed, our evaluation (Section 3.4) shows that the behavior signature is accurate.

Making such a security analysis work on a large number of apps, however, turns out to be challenging. Although many existing tools [14, 26, 27] already support a taint analysis on Android, all of them are designed to analyze the whole program, which is complicated and slow. The problem comes from the difficulty in building an app's whole CFG across IPCs: the construction of the CFG from an entry point has to stop at an IPC call and waits for an analysis on the call's parameters to determine which handler within the program will be invoked next. Running this on a complicated app is time consuming and also unnecessary, since all we need is just a quick analysis on a small portion of the code related to the push-messaging integration. Here we describe an *integration-specific* technique to improve the performance of the analysis.

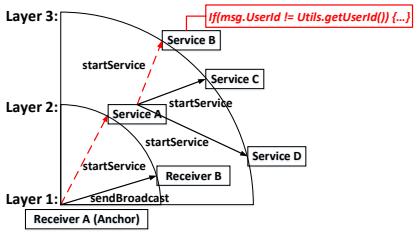

Figure 6: Example of Breadth-first and Layered Analysis

Specifically, our integration scanner performs a breadth-first, layered analysis. Starting from the anchor, it first builds up a partial CFG of the app, whose paths all end at the sites of IPC calls, as illustrated in Figure 6. Over this partial CFG, which is considered to be the first "layer" of the analysis, we run a taint check, using the function provided by FlowDroid [14]. For all the equality comparisons discovered in this way, our scanner evaluates whether any of them is a security check on user ID, as described above. If none of them matches the behavior signature, we select a set of paths on the partial CFG to extend to the next layer, that is, propagating the taint to their IPC handlers. These paths are selected when their end points, i.e, the IPC calls, are either `startService` or `sendBroadcast` (or `sendOrderedBroadcast`), as other IPC calls like `startActivity` are unrelated to the message processing. Also, these calls should have their Intents tainted, indicating that their handlers work on the message. The corresponding handlers are discovered using the names of the actions or classes specified in the calls.

Over the extension of the CFG on the next layer, our scanner continues to propagate the taint and evaluate the tainted equality comparisons against the signature. If none has been found, we do this again and extend the partial CFG to the third layer. In the case that the security check is not present even on this layer, the app is reported to fail to check User ID. This is because the User ID of the message needs to be evaluated early in the processing: if this has not been done by the `onReceive` of the anchor (the first layer) as soon as the message comes in, it must happen right after the format checking and classification either within the `onReceive` or in a service the anchor invokes (the second layer); therefore a three-layer analysis is sufficient for determining whether such a check indeed exists. In this way, our scanner only needs to go through a small portion of app code, allowing it to run much faster than a direct use of existing tools.

3.4 Evaluation

We implemented Seminal on top of FlowDroid [14], and ran it on the integrations of 30 push-messaging services within 35,173 apps downloaded from 6 markets. We discovered 17 security-critical flaws within 30 SDKs, and 26,069 potential problems in 17,668 apps, including high-profile ones such as Facebook, Skype, Yelp, etc. The detailed findings are elaborated in Section 4. Here we report how our system performed.

Effectiveness. In the experiments, our implementation accurately evaluated all the sample code and related manifests. When it comes to the 30 SDKs, the prototype reported that 14 of them contain inse-

cure broadcast channels (unprotected receiver or broadcast without target package) along the execution paths for message delivery, and the rest 16 do not have such a problem. We manually verified these findings, which were all confirmed to be correct.

For the CID, we found that 10 out of 30 SDKs do not generate the secret identifier at all. Instead, some of them only use `tag` to label a group of users for broadcasting messages to them, and the others ask the developer or the user to come up with an *alias* (e.g., email address) for identifying the app, which itself has security risks (Section 4). Among the rest 20 SDKs, our prototype successfully went through 15 of them. The rest 5 could not be analyzed because their code related to CID generation and processing has been obfuscated or contains extremely complicated data structures. Note that this does not undermine the utility of Seminal, as the developer who wants to use it for vulnerability detection will not deliberately obfuscate her code. Among those our prototype can handle, 7 were found to build their CID either using Android's UUID generator or with the input from the connection server, which are likely to be secure. Also, Appsfire was found to directly use the GCM registration ID. The remaining 7 turned out to be problematic: 5 of them either solely rely on public resources (e.g., `android_id`) for constructing the secret token or inadvertently expose it to unauthorized parties; the remaining 2 was found to build their CID partially from public resources but our prototype could not determine whether non-public data are also involved, due to the complexity of their program structures. We elaborate these problems in Section 4.1. Such findings were validated through manual inspections.

Using the database generated at the first two stages, our system scanned over 35,173 apps integrating push-messaging services, 17,616 from Google Play Store and the rest from third-party markets. Among the apps reported by Seminal as vulnerable, we found that all the findings are accurate in the cases of insecure IPC receivers, `PendingIntent` and other authentication issues. However, for User-ID checks, a small set of cases turned out to be false positives: that is, the apps actually performed User-ID filtering while our implementation failed to identify the presence of such an operation. Specifically, we inspected 12 most popular apps in the social category that were found to have the User-ID problem, manually evaluating their code and also performing a dynamic analysis to validate the presence of the vulnerability. 10 of them were confirmed to have the security flaw and 2 were false positives. A close look at these false positive cases reveals that one of them was actually caused by the taint analysis mechanism provided by FlowDroid, which missed the equality check present in the code. It is important to point out that these apps are among the most complicated ones we analyzed and the real false detection rate is much lower: in another validation attempt, we *randomly* sampled 20 apps across all categories and only found one false positive, which was caused by FlowDroid [17, 27].

Performance. Our implementation of Seminal was found to be efficient. For the SDK it can handle (particularly the one without deep obfuscation), the prototype always completed the analysis within 10 minutes. When it comes to the security analysis that happened on apps, which includes User-ID filtering, protection of anchor receivers and exposure of `PendingIntent`, on average 108 seconds were spent on each app. It is important to note that such a performance is achieved by our unique design that automatically identifies and analyzes only part of app code related to service integration.

4. NEW ATTACKS AND MEASUREMENT

In this section, we report our findings, focusing on new security flaws and interesting observations.

SDK and app collection. As mentioned earlier, the services we studied include popular syndication services and third-party services whose SDKs were publicly available. All the syndication services here are provided in North America, while all the third-party services are from China, where major manufacturer services such as GCM are not accessible. Also there are tens of services we were not able to study, due to the difficulty in obtaining their SDKs, which requires application and approval, e.g., IBM Xtify. Some new services have not yet publicly released their SDKs (e.g., Mono Push). This demonstrates that the market of push messaging is highly vibrant and therefore developing effective means to ensure security qualities of the services is imperative.

Our implementation also fingerprinted popular manufacturer services, including GCM, ADM and Nokia. Those services mainly rely on their client-side service apps such as `android.gms` to deliver messages. In addition, GCM and ADM were manually analyzed by the prior work [28]. Therefore in our research, we just focused on these services' integrations on the app side.

Market	No. Of Apps
Google Play	17,616
Baidu	2,315
Gfan	3,734
Appchina	4,572
Mumayi	3,612
Xiaomi	3,324
Overall	35,173

Table 2: Sources of Analyzed Apps

17,616 of the 35,173 apps used in our study come from Google Play, which include those integrating push-messaging services and also highly ranked in each category (social, health, finance, etc.), and the rest from 5 popular app markets in China, which use third-party services. Table 2 summarizes the sources of these apps. Among them are highly popular ones such as Facebook, Skype, Yelp, Pinterest, etc., with 1682 having over 1 million downloads.

4.1 New Attacks

The new problems discovered mainly come from the confusion caused by vulnerable service integrations: an app can be confused by where the message it receives comes from when it integrates multiple services, by who it is supposed to deliver the message to and by which service app it should talk to. Also, the CIDs used by syndicators or third-party services are often not kept secret.

Service confusion. Seminal is designed to check whether an SDK and its hosting app properly authenticate the senders of the Intents they receive (Section 3.2). When the receivers of these IPCs are found to be protected by permissions, it becomes important to know that such permissions are only given to the right party, which is almost always either the hosting app itself or a system app. Therefore, whenever our implementation scans an app or an SDK, it always reports the presence of the permissions, asking for the confirmation that they are indeed defined and claimed by authorized parties. In our study, our implementation discovered that 51 popular apps, most with over 1 million downloads, and 2 SDKs (Urban Airship and PushIO) integrate multiple push-messaging services, including GCM, ADM and Nokia. The outputs of the analysis are a set of permissions (e.g., `com.google.android.c2dm.permission.SEND` for GCM, `com.amazon.device.messaging.permission.SEND` for ADM, `com.nokia.pushnotifications.permission.SEND` for Nokia), which are all required to be given to system apps only.

The problem, however, is that this condition cannot be satisfied in practice. Most of Android devices only support one of such services: that is, only one service app (`android.gms` on Nexus or `com.amazon.device.messaging` on Kindle) exists on a given device, defining only one of such permissions. As a result, the other permissions are up for grabs by any parties: any app can define such a permission within its manifest to gain the capability to send messages to the target app's receiver it protects. For example, on Kindle Fire, a malicious app can define `com.google.android.c2dm.permission.SEND`, the permission for the GCM service `android.gms`, which is not running on the device; as a result, the adversary becomes able to send any messages to the apps integrating both the ADM and the GCM services, either directly or through an SDK.

In our study, we found that 51 popular apps have this problem. Particularly, on Kindle Fire, we built end-to-end attacks in which an attack app successfully impersonated authorized parties (e.g., close friends of the victim user) to inject messages to the victim's Facebook and Skype apps. On the Nexus device, the same trick can also be played: the attack app can define the ADM permission `com.amazon.device.messaging.permission.SEND` to send messages to the apps. However, the attacks can only cause a denial of service, crashing both apps. A close look at the problem reveals that when processing the messages, the integrations within both apps ask for some ADM objects not present on Nexus, which leads to the crash[3]. However, we found that when the attack app defined the Nokia permission, `com.nokia.pushnotifications.permission.SEND`, on Nexus, whose service has also been integrated into the Facebook app and Skype, it successfully injected messages and made the apps display them to the user in her friend's name (demo [11]). All together, our scanner reported 382 apps having this vulnerability, including the 51 with over 1 million downloads.

Even more serious here is that both Facebook and Skype, and most likely many other apps with the same problem are also subject to a message stealing attack. Specifically, we found that on Kindle Fire, a malicious app defining the GCM permission can impersonate a non-existing GCM service app to inject a *GCM registration ID*, which has been bound to an attack device, into the Facebook or Skype app. As a result, the apps will be cheated into believing that they are actually using the GCM service (instead of ADM), and accordingly send the registration ID to their GCM app servers[4] to bind the current user to that ID. The consequence is that from that point on, only their GCM app servers can push messages to these apps and the messages actually all go to the attack device, because the registration ID here is tied to the device. In our research, we successfully launched the attack on Kindle Fire, and received push notifications from both Facebook and Skype, including such sensitive information as Skype's private messages. The same security risk is also present on Nexus phones, when a malicious app impersonates the Nokia service app.

The same problem (service confusion) also happens to the SDKs of popular syndication services, including Urban Airship [12], which recently incorporates both GCM and ADM. Particularly, we found that PushIO only maintains one registration ID, no matter whether it comes from GCM or ADM. As a result, a malicious app can replace the target app's GCM ID using the ADM permission on Nexus, by sending the new ID to the target's ADM receiver (within the SDK). The same attack also succeeded on Kindle Fire, by replacing an app's ADM ID through its GCM receiver. Note that this is slightly different from our attacks on Facebook and Skype, in which the adversary stealthily binds the apps from the ADM service to GCM. Here what we can do is directly replacing the SDK's GCM registration ID through the ADM channel. This vulnerability was acknowledged by related organizations as a serious security flaw. Facebook rewarded us $2000 for helping them fix the problem.

User confusion. When scanning all 35,173 apps, we deliberately selected those involving user-related sensitive information to find out whether they properly check the User ID of an incoming message. This was done by focusing on the apps from the categories such as social, finance, health, etc., and inspecting their user interfaces for the keyword "password", which almost always indicates the presence of users' login credentials. Our analysis identified that popular apps like Yelp, Pinterest, etc., did not perform any user-ID based access control.

To understand the security consequences of this user confusion problem, we built end-to-end attacks to exploit high-profile popular apps. The idea is to associate the target app to a different user, the adversary, so that he can push messages to the victim through his own account on the target's app server (e.g., the Pinterest server). Such an association can be established when the adversary gets his hands on the target app's registration ID or CID. It is important to note that in the absence of the user-ID vulnerability, knowing the registration ID does not enable the adversary to do anything harmful: he cannot get access to the legitimate user's messages, as the registration ID is *not* an authentication token and also bound to the target app; nor can he inject any messages to the app because of its check on user IDs, which filters out the messages from a different user (the adversary in this case). For CID, its leakage indeed allows the adversary to access the victim's messages. However, he still cannot inject messages to the target app when it verifies user ID.

In our research, we exploited the security flaw within Pinterest, a popular social app. What we did is to leverage a known vulnerability in the browser and webview's origin-based protection to acquire the GCM registration ID within the app [4]. The vulnerability here was found still pervasive among the devices with Android 4.3 and below, which are running on about 69.8% of Android devices [1]. Specifically, whenever the Pinterest app visits a malicious web site, the adversary can open an iframe within its webview instance to upload the app's `com.google.android.gcm.xml`, which contains the registration ID. The content within the iframe can be acquired by the script injected, due to a weakness in webview's input sanitization mechanism [4]. In this way, the adversary can get the registration ID from the app and further send it to the Pinterest server to bind the ID to the attacker's account there. Such an attack was confirmed in our research, allowing us to push messages to our own accounts but make them delivered to the target app (the victim's Pinterest app). We also built a similar attack on the Yelp app, using another known vulnerability CVE-2012-6636 [3] to get the victim's registration ID before exploiting Yelp's lack of user-ID verification to push fake messages to the victim. In addition, for all the apps integrating the SDKs that leak out CIDs, such as Push Woosh, the attacker can easily bind their CIDs to his account so as to push messages to the victim. This threat was also found to be practical in our study.

Apparently, a solution to this problem is a security check on the app server side to make sure that no two users share the same registration ID or CID. Indeed, we observed that some app servers, such as Ask.fm, Lovoo.com detach the registration ID from one user and binds it to another whenever the former logs out from their apps and the latter logs in. However, this treatment cannot stop

[3]Kindle Fire actually includes the Android objects GCM needs.

[4]Actually, for those big organizations, they typically have their own syndication services that connect to both GCM and ADM.

the user confusion attack: knowing the victim's registration ID, the adversary can simply log into the target app on his device to make the app server bind the ID to his user account. When a message is pushed to the ID, however, the victim will still be the recipient because GCM ties the ID to her device. In this way, the adversary still can inject messages to the victim's app, in the name of the victim's friend. Fundamentally, the problem is caused by the fact that the push-messaging cloud can only identify the app on a specific device, not the user of the app. Therefore, as long as an integration fails to check user IDs, the user will always be under the risk of receiving messages from an untrusted party.

Client confusion. Also our prototype reported that 11,841 apps we scanned have insecure broadcast channels (unprotected receiver or broadcast without target package). For example, most of the receivers for getting the messages from the service SDKs are completely unprotected and therefore any apps running on the same device can directly inject messages into these vulnerable apps. Although in the most cases, the problem is apparently caused by implementation errors, there are situations when the security implication is more fundamental. A prominent example here is the SDK provided by Baidu Cloud Push, one of the most popular push-messaging services in China with hundreds of millions of users [2]. Our analysis on their vulnerable receivers (detected by Seminal) reveals the security issues in their designs.

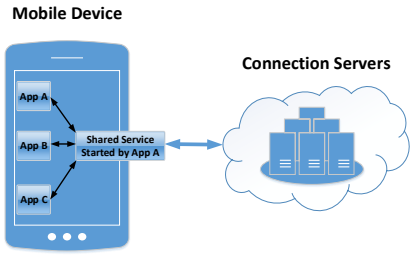

Figure 7: Push-Messaging with Shared Service

It turns out that some services (e.g. Baidu) adopt a strategy to support multiple hosting apps running on the same device, apparently for the purpose of power saving. Specifically, in the presence of these apps, one of them will launch a process to serve *all* of them: the process directly communicates with the connection service in the cloud to acquire messages for all these apps, and then pushes the messages to their recipient apps' receivers (based on their package names). Figure 7 shows how this mechanism works. Since these apps can come from any parties, as long as they integrate the SDKs of the services, the permission-based protection is no longer applicable here, as it is hard to determine which app should get the permission and which should not.

The consequence of the problem is that in addition to the message injection threat, all these services may be subject to a man-in-the-middle (MitM) attack. There is nothing to prevent a malicious app from claiming that it also integrates the services, and therefore becoming entitled to launch the service process. The process is trusted to relay the messages for other apps pushed from the cloud, and therefore is well positioned to collect the app users' messages and even modify their content. Further, some of these services utilize a volunteer mechanism to coordinate their customers' apps: each app can set its own priority, and the one with the highest value is supposed to create the service process while the others are asked to terminate their processes to save battery power. This design enables a malicious app to become the MitM whenever it wants. In our research, we built an end-to-end attacks on Baidu Cloud Push and successfully intercepted legitimate apps' messages, which indicates that the problem is indeed serious.

CID exposure. Our Seminal scanner also discovered the pervasiveness of CID exposure: among all 22 services designed to push messages to individual users through CIDs, 7 turned out to either create the identifiers using publicly available resources (also accessible to malicious apps) or expose their content to the unauthorized parties. As an example, we found that high-profile services like Push Woosh actually utilize *deviceID* and *android_id* to build their CIDs. Such information can be acquired by a malicious app on the victim's devices using the permissions like `READ_PHONE_STATE`. With such information, we were able to generate the same CIDs for apps using such services on an attack device. Since the CIDs serve as authentication tokens between the devices and the connection server, we successfully utilized them to obtain the target apps' messages on the attack device.

Of particular interest here is the syndication service PushIO, which generates its CID using the Android `uuid` API, a rather secure approach. This CID is supposed to be bound to the GCM (or ADM or other manufacturer cloud's) registration ID for the app integrating the service to enable a message push through the identifer, regardless what manufacturer service the app is using. When the GCM registration process (for getting the registration ID) fails, which happens from time to time, the app needs to try it again. This retry operation is initiated by the SDK through sending an Intent to its hosting app. The way to do this, however, is through a broadcast, with the CID included in the Intent for the recipient to find out whether it is the right party to act, in the case that multiple apps using this service are present on the same device. The problem here is that a malicious app can register the same action the legitimate app's receiver uses to get the CID within the Intent.

Other security risks. As discussed before, 10 out of the 30 SDKs we studied do not use CID or registration IDs. Most of them apparently are designed to push messages to a group of users, instead of a single individual. However, some services (e.g. YunBa) also provide an *alias* mechanism for locating a specific app. The alias here is an identifier created by the developer or the user, just like a user ID and password. The problem is that the documentations of these services suggest to use public information like one's email address for this purpose. As a result, the adversary can easily figure out the alias and use it to access the messages for the target apps.

4.2 Measurement and Discoveries

Risk	Analyzed Apps	Vulnerable Apps
Service Confusion	17,616	382(2.17%)
User Confusion	3086(8 sensitive categories)	2,234(72.39%)
PendingIntent	17,616	2,101(11.93%)
Overall	17,616	4,368(24.80%)

Table 5: GooglePlay App Risks

Risk	Analyzed Apps	Vulnerable Apps
Insecure Broadcast Channel	17,557	11,841(67.44%)
User Confusion	17,557	5,436(30.96%)
PendingIntent	17,557	4,075(23.21%)
Overall	17,557	13,300(75.75%)

Table 6: Chinese Markets App Risks

Landscape. From all 35,173 apps integrating push-messaging services, Seminal found that 17,668 of them, more than 50%, have different security weaknesses. Table 5 and Table 6 provide the breakdowns of the findings. For all the apps from the Chinese markets

Downloads	Service Confusion	User Confusion	PendingIntent	Overall
1-100	70/837(8.36%)	162/196(82.65%)	64/837(7.65%)	279/837(33.33%)
100-10K	159/6864(2.32%)	1132/1453(77.91%)	707/6864(10.30%)	1843/6864(26.85%)
10K-1M	102/8233(1.24%)	811/1143(70.95%)	1085/8233(13.18%)	1849/8233(22.46%)
1M-100M	47/1646(2.86%)	126/282(44.68%)	245/1646(14.88%)	391/1646(23.75%)
100M+	4/36(11.11%)	3/12(25%)	0/36(0%)	6/36(16.67%)

Table 3: GooglePlay App Vulnerability Based on Number of Downloads

Category	Service Confusion	User Confusion	PendingIntent	Overall
Medical	16/277(5.78%)	157/184(85.33%)	50/277(18.05%)	181/277(65.34%)
Shopping	38/676(5.62%)	383/497(77.06%)	116/676(17.16%)	449/676(66.42%)
Business	33/1027(3.21%)	605/790(76.58%)	151/1027(14.70%)	724/1027(70.50%)
Health & Fitness	10/455(2.20%)	258/359(71.87%)	46/455(10.11%)	291/455(63.96%)
Social	13/719(1.81%)	349/574(60.80%)	90/719(12.52%)	395/719(54.94%)
Finance	8/533(1.50%)	285/371(76.82%)	87/533(16.32%)	333/533(62.48%)
Personalization	NA	21/37(56.76%)	3/320(0.94%)	23/320(7.19%)
Communication	5/381(1.31%)	176/274(64.23%)	49/381(12.86%)	207/381(54.33%)

Table 4: GooglePlay App Vulnerability Based on Category

(Baidu, Gfan, Mumayi, Appchina and Xiaomi), 75.7% are vulnerable, with the leading cause of insecure broadcast channels (unprotected receiver or broadcast without target package). Among all the Google-Play apps, 24.8% are problematic, with most of them suffering from the lack of User-ID checks. It is interesting to see that the geographic split also brings in a discrepancy in the types of vulnerabilities among those apps: Chinese apps are completely free from the service confusion problems (e.g., integration of both GCM and ADM), as they typically just incorporate a single service, while only a small set of apps on Google Play have the IPC problems. When it comes to push-messaging services, Table 7 shows that also more than half of them are vulnerable. Particularly, most service SDKs in China have security weaknesses, the main cause for the pervasiveness of vulnerable apps there. 4 syndication services also contain different kinds of problems.

Consequences. When we look at the categories of the apps that tend to be problematic, as illustrated in Table 3, it is alarming to see that all those with sensitive user data appear on the top. 50 - 70% of the Google-Play apps for business, shopping, medical, health, finance, communication and social are found to be riddled with loopholes, mostly caused by missing User-ID checks (55 - 85%). Also 10 - 18% of these apps expose `PendingIntent`, which essentially allows a malicious app to inject any content to their receivers and intercept all of their users' private messages through replacing their registration IDs with that of an attack device. In addition, over 5% of medical and shopping apps have the service confusion problem, vulnerable to message injection from the malicious apps that impersonate the messaging service app not on the victim's device. Note that this ratio is higher than the average (2.17%).

The consequences of the attacks are dire. Once successfully exploiting target apps' User-ID or service-confusion weaknesses, the adversary can fake messages and post them to the victim's apps (Pinterest, Yelp etc.). Also through the `PendingIntent` bound to the attack device, private user information like chat messages, financial information, etc. will be delivered to the adversary. For tens of thousands of apps in the Chinese markets, their vulnerable IPCs may enable both injection of misleading messages and disclosure of confidential user data. Table 8 presents examples of the information assets at risk in the presence of successful attacks.

Impact and trend. The impacts of the vulnerabilities we discovered are significant: from Table 4, we can see that over 23% of the popular apps (each with more than 1 million downloads) are vulnerable, which includes 6 (such as Facebook, Skype) being installed over 100 million times. Also interestingly, the service confusion problem tends to occur in the apps either extremely popular (above 100 million downloads) or no one uses (below 100 installs). Specifically, over 3% of the apps with more than 1 million downloads have the problem, in contrast to 1.7% among those with 100 - 1 million downloads. This could be explained by the observation that those popular apps are more likely to integrate multiple push-messaging services and therefore more exposed to this security risk. In the meantime, over 8% of the least popular apps also contain the security flaws. It turns out that they were all built on the templates developed by the companies like app4mobile and conduit, which integrate multiple push-messaging services.

Also we found that popular apps are more likely to disclose `PendingIntent` than less popular ones, as illustrated in Table 4. Such disclosure almost always happens to those using the vulnera-

Service	Type	Weaknesses
Urban Airship	Syndication	Service Confusion
PushIO	Syndication	Insecure Broadcast Channel/ CID Exposure/Service Confusion
Push Woosh	Syndication	Insecure Broadcast Channel/ CID Exposure
Pushapps	Syndication	CID Exposure
Baidu	Third-Party	Insecure Broadcast Channel
Getui	Third-Party	Insecure Broadcast Channel
Xiaomi	Third-Party	Insecure Broadcast Channel
XG Push	Third-Party	Insecure Broadcast Channel/ CID Exposure
Bmob	Third-Party	Insecure Broadcast Channel
Yunba	Third-Party	Insecure Broadcast Channel
Zhiyou	Third-Party	Insecure Broadcast Channel
Mpush	Third-Party	Insecure Broadcast Channel/ CID Exposure
LeanCloud	Third-Party	Insecure Broadcast Channel
Umeng	Third-Party	Insecure Broadcast Channel/ CID Exposure(risk)
JPush	Third-Party	Insecure Broadcast Channel
Shengda Push	Third-Party	CID Exposure
Huawei	Third-Party	Insecure Broadcast Channel

Table 7: Push-Messaging Services Weaknesses

App	Downloads	Vul type	Sample Contents at Risk
Facebook	500M+	Service Confusion	Messages
Skype	100M+	Service Confusion	Messages
Pinterest	10M+	User Confusion	Messages
Yelp	10M+	User Confusion	Messages
Linkedin	10M+	PendingIntent	Invitation, Messages
eBay	50M+	PendingIntent	Shipment, Messages

Table 8: Examples of Vulnerable Popular Apps

ble GCM template [5]. Apparently, Google needs to do more to fix this problem. The only flaw distributed more in line with what is expected is the User-ID confusion: the more popular the apps are, the less likely they have this issue. On the other hand, as discussed before (Section 4.1), still high-profile apps like Pinterest (10M+ downloads) and Yelp (10M+ downloads) contain this flaw.

5. RELATED WORK

Security analysis of push-messaging services. Little has been done on evaluating the security qualities of push-messing services until very recently, when researchers took a close look at the GCM, ADM, UrbanAirship and a Chinese Push Cloud [28]. What have been found include the security-critical vulnerabilities in GCM's cloud-side security checks and a few client-side problems mostly caused by exposure of `PendingIntent`. Different from this prior work, which is based upon manual analysis of a small set of services and apps, our research is much deeper and broader: we built an automatic analysis tool and ran it to scan 30 popular cloud-messaging services and 35,173 popular apps; we found 17 vulnerable SDKs and 17,668 faulty apps (including high-profile ones such as Facebook, Skype, Baidu, etc.), with a significant portion of them involving *previously unknown* security flaws.

Android ICC security. The security issues of Android IPC or more precisely Inter-Component Communication (ICC) have been extensively studied. Comdroid [15] and other work [18, 17] investigate the Intent-based attack surface, including the insecure broadcast channel issue. Other prior work identifies different kinds of risks in the ICC channel, including information leaks and pollution in content provider [38], permission re-delegation problems [20] and capability leaks [29, 23]. Different from these prior studies, what we want to understand here is the security implications of the ICC vulnerabilities to push-messaging services. For this purpose, we developed a unique analysis technique, which utilizes the anchor discovered from sample code to backtrack all the ICC calls that pass the message received by an SDK to its hosting app and inspect their security settings. Also, the understanding of the unprotected receivers and other ICC interfaces has also been put in the context of push-messaging service: for example, our study reveals a design flaw in Baidu Cloud Push (Section 4.1), which actually deliberately exposes an app's ICC receiver for coordinating different apps integrating its SDK. When it comes to `PendingIntent`, prior research briefly mentions the lack of origin information within the Intent mechanism [33], which forces the app developer to utilize `PendingIntent` to provide the origin of an ICC request. Our prior research [28] first demonstrates the security impacts of such exposures on push-messaging services. However, never before has any effort been made to automatically detect such a vulnerability, as we did in our research. Furthermore, the service confusion problem has never been studied before: unlike the prior work that focuses on unprotected components, here we look at the situation where the permissions for protecting these components are completely missing in a system.

Static analysis on Android. How to statically analyze Android system components and app code has been studied for years. Many tools have been built to identify vulnerabilities [15, 29, 16, 19, 25, 32, 31], privacy leaks [21, 37, 36, 22, 34, 35] and malware [39, 13, 24]. Particularly, many systems are designed to detect ICC vulnerabilities. For example, Comdroid [15] detects multiple intent related vulnerabilities. Chex [29] analyzes permission, capability and privacy data leakage caused by unprotected components. ContentScope [38] detects the vulnerabilities in content providers. Woodpecker analyzes the ICC vulnerabilities in the preloaded apps from firmware. Compared to these prior studies, our work focuses on ICC vulnerabilities in push messaging service SDKs and their integrations within apps, and proposes novel sample code guided analysis techniques to detect unprotected ICC interfaces. Further, data flow analysis is widely used in static analysis of mobile applications. Particularly, Amandroid [34] constructs the Inter-component Data Flow Graph (IDFG) and data dependence graph (DDG) to detect data flow. DroidSafe [22] builds a model for Android runtime named Android Device Implementation (ADI) and performs an information flow analysis using the model. FlowDroid [14] is a precise static taint analysis tool for Android apps, which we utilized to build Seminal. Its problem is the limited capability to handle ICC (IPC as discussed in the paper). ICC inference (for linking call sites to their handlers) is the focus of Epicc [30], which has been combined with FlowDroid in the systems like Didfail [26] and IccTA [27] to enable both inter-component and intra-component data flow analysis. However, these generic data flow analysis tools aim at evaluating the whole app, instead of just the integration part we are interested in. As a result, they are not efficient for our purpose. By comparison, the unique design of Seminal, particularly the anchor-based, layered security analysis (Section 3.3), making it potentially much more efficient than those generic tools when analyzing push-messaging integrations.

6. DISCUSSION

Running Seminal on our app collection, we discovered tens of thousands of vulnerable apps. The presences of security risks within these apps have been validated through a manual analysis on randomly selected samples. This, however, does not mean that all these apps can be exploited. A successful attack on a known vulnerability often depends on other issues such as availability of message formats (for message injection). Sometimes, we may even need to leverage multiple flaws to make an end-to-end attack work. Given the large number of problematic apps we discovered, it is impossible for us to exploit even a small portion of them within a short period of time. Therefore, except a few high-profile apps we indeed broke, the other findings are security risks in a strict sense. Nevertheless, all such problems are indeed security critical, rendering the apps much more exposed to security threats than those that do not have them.

The techniques we developed and the study we performed are nothing more than a first step towards effective protection of push-messaging services. After all, we only looked at the service integration between app and SDK. It is still less clear whether other security problems exist on the cloud front. This can be addressed by formal verification of the whole service. To this end, new techniques need to be developed for probing the cloud-side program logics to build a more accurate model. Also, the rapid evolution on the messaging-service infrastructure requires continuous studies to identify its new security-related features.

7. CONCLUSION

In this paper, we report the first large-scale, systematic security analysis on the integration of push-messaging services within Android apps. This study was performed using Seminal, a new tool for automatically verifying key security properties in such integrations. Seminal is designed to leverage unique features of push-messaging services, focusing on only a small portion of the app code related to the integration. Running it on 30 leading cloud-messaging services and over 35,173 popular apps, we are amazed by the scope and the magnitude of the security problem in those services: more than half of the service SDKs, over 20% of Google-Play apps and more than 50% apps from mainstream Chinese markets involve critical secu-

rity risks (see Section 6), allowing the adversary to impersonate trusted parties to post content to the victim's apps and intercept her private messages. More importantly, most of the flaws are never known before. Our findings indicate the seriousness of the problem, which requires intensive effort to address, and our new technique makes the first step toward improving the security qualities of this important mobile cloud service.

8. ACKNOWLEDGMENTS

The project is supported in part by National Science Foundation CNS-1117106, 1223477 and 1223495. Authors from Peking University are supported in part by National Development and Reform Commission (NDRC) under project "Guidelines for Protecting Personal Information". Kai Chen was supported in part by NSFC 61100226.

9. REFERENCES

[1] Android Platform Distribution. https://developer.android.com/about/dashboards/index.html.

[2] Baidu Cloud Push. http://developer.baidu.com/cloud/push.

[3] CVE-2012-6636. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-6636.

[4] CVE-2014-6041. http://nvd.nist.gov/nvd.cfm?cvename=CVE-2014-6041.

[5] GCM Template Code. http://developer.android.com/google/gcm/c2dm.html.

[6] Getui. http://www.igetui.com/.

[7] JPush. https://www.jpush.cn/.

[8] Push Woosh. https://www.pushwoosh.com/.

[9] PushIO. http://www.responsys.com/marketing-cloud/products/push-IO.

[10] Soot. http://www.sable.mcgill.ca/soot/.

[11] Supplement materials. https://sites.google.com/site/perplexedmsg/.

[12] UrbanAirship. http://urbanairship.com/.

[13] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2014.

[14] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 29. ACM, 2014.

[15] 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*, pages 239–252. ACM, 2011.

[16] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 73–84. ACM, 2013.

[17] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *USENIX Security Symposium*, 2011.

[18] W. Enck, M. Ongtang, P. D. McDaniel, et al. Understanding android security. *IEEE Security & Privacy*, 7(1):50–57, 2009.

[19] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: An analysis of android ssl (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 50–61. ACM, 2012.

[20] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, 2011.

[21] C. Gibler, J. Crussell, J. Erickson, and H. Chen. *AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale*. Springer, 2012.

[22] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. Information-flow analysis of android applications in droidsafe. In *Proc. of the Network and Distributed System Security Symposium (NDSS). The Internet Society*, 2015.

[23] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, 2012.

[24] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 281–294. ACM, 2012.

[25] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 66–77. ACM, 2014.

[26] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6. ACM, 2014.

[27] L. Li, A. Bartel, T. F. D. A. Bissyande, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE 2015)*, 2015.

[28] T. Li, X. Zhou, L. Xing, Y. Lee, M. Naveed, X. Wang, and X. Han. Mayhem in the push clouds: Understanding and mitigating security hazards in mobile push-messaging services. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 978–989. ACM, 2014.

[29] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240. ACM, 2012.

[30] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *USENIX Security 2013*, 2013.

[31] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious

dynamic code loading in android applications. In *Proceedings of the 20th Annual Network & Distributed System Security Symposium (NDSS)*, 2014.

[32] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps. In *Proceedings of the 19th Network and Distributed System Security Symposium*, 2014.

[33] R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 635–646. ACM, 2013.

[34] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1329–1341, New York, NY, USA, 2014. ACM.

[35] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu. Effective real-time android application auditing. In *IEEE S&P*, 2015.

[36] Z. Yang and M. Yang. Leakminer: Detect information leakage on android with static taint analysis. In *Software Engineering (WCSE), 2012 Third World Congress on*, pages 101–104. IEEE, 2012.

[37] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054. ACM, 2013.

[38] Y. Zhou and X. Jiang. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Annual Symposium on Network and Distributed System Security*, 2013.

[39] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, 2012.

APPENDIX

Models and Security Properties

To analyze the security qualities of different push-messaging services, we first need to find out the security properties they are expected to have. In this section, we describe how we model these services and identify their necessary properties, with a focus on their integrations within apps. Understanding the integrations is a critical first step towards evaluating a push-messaging service's security quality, as the prior research [28] clearly indicates that the weakest link of security protection is on the mobile device. Also, due to our limited observation of what happens within the cloud (e.g., the program logic of the connection server), a study on complete services is hard. Following we describe the model and properties. All the information used here comes from the specifications of existing messaging services.

Models. The security goals of a push-messaging service can be described as follows: (1) a message m_u for a user u from an app server $appsrv$ should only be pushed to u and no one else; (2) u only gets her own message m_u (not others) from the authorized server $appsrv$ (not from an unauthorized party). The security policies for achieving these goals need to be enforced by the reference monitors distributed across different entities within a push-messaging service. Specifically, such a service can be modeled as $\langle S, A, I\rangle$. Here S is a set of *states* where a message or a service request is being processed by a service entity, such as the state of app server ($appsrv$), connection server ($connsrv$), syndication server ($synsrv$), device-side service app ($client$), SDK (sdk) or app (app). A is a set of security-related *actions* that happen on a state. For example, $A(connsrv) = (sreceive(i, appsrv \rightarrow connsrv), check(i), ssend(convert(i), connsrv \rightarrow client))$ models the operations performed by a connection server (at the $connsrv$ state) given an input i from an app server: $sreceive$ receives i from the network and authenticates its sender; $check$ verifies i against a security policy that only an authorized sender (the one bound to the registration ID in i) is allowed to push a message to the ID; $convert$ transforms i to i' by replacing the registration ID with the target device and app, and then $ssend$ sends the input to the service app (across the network) through an authenticated secure channel and also causes the system to move to the next state $client$. Finally I is the collection of inputs to those states, in which i is a concatenation of identifier (dev), authentication tokens (e.g., $appid$) and message (m).

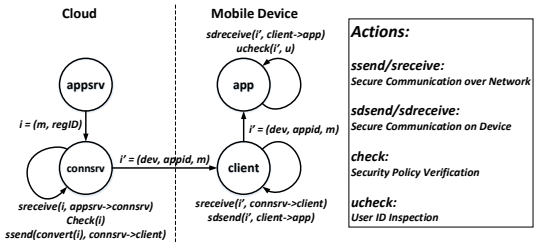

Figure 8: Manufacturer Push Service Security Model

Using this simple model, Figure 8 illustrates the security checks expected when a message is pushed to the user through the manufacturer service. Specifically, for the manufacturer service, after the input $i' = (dev, appid, m)$ is sent to $client$, the service app continues to perform $(sreceive(i', connsrv \rightarrow client), sdsend(i', client \rightarrow app))$, where $sdsend$ passes a message *on device* to another Android component through an authenticated secure channel. Intuitively, these actions include authenticating the connection server and sending i' to the target app. Then, app further goes through $(sdreceive(i', client \rightarrow app), ucheck(i', u))$, where $sdreceive$ verifies the sender on the same device and $ucheck$ inspects the user ID within i' against that of the current login user u to make sure that she is the right recipient. Note that such a verification can only happen within the app, because no other entities in the service can differentiate two users sharing the same app on the same device, which if not handled properly can lead to serious security breaches (Section 4.1).

For a third-party service, the reference monitor on the connection server $connsrv$ performs exactly the same operations as described above, except that $ssend$ at $connsrv$ sends i' directly to sdk through a socket connection. Here $ssend$ uses the CID generated by the SDK during the registration process to establish such a secure channel. In the state sdk, actions $(sreceive(i', connsrv \rightarrow sdk), sdsend(i', sdk \rightarrow app))$ are taken to pass i' securely to the app, which further checks the user information within the input ($ucheck$). When it comes to the syndication service, an additional state $synsrv$ needs to go through during message pushing, where a function $sconvert$ replaces the CID within i (from the app server) with the registration ID of a manufacturer cloud. Also on the device side, the $client$ state moves to sdk before arriving at app. Along the transitions, authentication needs to be performed at every state.