Vulnerable Implicit Service: A Revisit

Lingguang Lei[¶†‡], Yi He[§], Kun Sun[‡], Jiwu Jing[¶†], Yuewu Wang[¶†], Qi Li[§], Jian Weng[*]

[¶]Data Assurance and Communication Security Research Center, Chinese Academy of Sciences, China
[†]State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences
[‡]George Mason University, Fairfax, VA, USA
[§]Graduate School at Shenzhen, Department of Computer Science, Tsinghua University, China
[*]Jinan University, Guangzhou, China
leilingguang@iie.ac.cn,heyi14@mails.tsinghua.edu.cn,ksun3@gmu.edu,{jingjiwu,wangyuewu}@iie.ac.cn
qi.li@sz.tsinghua.edu.cn,169375@qq.com

ABSTRACT

The services in Android applications can be invoked either explicitly or implicitly before Android 5.0. However, since the implicit service invocations suffer service hijacking attacks and thus lead to sensitive information leakage, they have been forbidden since Android 5.0. Thereafter since the Android system will simply throw an exception and crash the application that still invokes services implicitly, it was expected that application developers will be forced to convert the implicit service invocations to explicit ones by specifying the package name of the service to be called.

In this paper, we revisit the service invocations by analyzing two sets of the same 1390 applications downloaded from Google Play Store before and after the the implicit service forbidden policy is enforced. We develop a static analysis framework called ISA to perform our study. Our analysis results show that the forbidden policy effectively reduces the number of vulnerable service invocations from 643 to 112, namely, 82.58% reduction. However, after a detailed analysis of the remaining 112 vulnerable invocations, we discover that the forbidden policy fails to resolve the service hijacking attacks. Among the 1390 applications downloaded in May 2017, we find 36 popular applications still vulnerable to service hijacking attacks, which can lead to the loss of user bank account and VPN login credentials, etc. Moreover, we find that the forbidden policy introduces a new type of denial of service attacks. Finally, we discuss the root challenges on resolving service hijacking attacks and propose countermeasures to help mitigate the service hijacking attacks.

CCS CONCEPTS

•**Security and privacy** →**Mobile platform security;** *Software security engineering;*

KEYWORDS

Implicit Intent; Service Hijacking Attacks; Denial of Service Attacks

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS'17, Oct. 30–Nov. 3, 2017, Dallas, TX, USA.
© 2017 ACM. ISBN 978-1-4503-4946-8/17/10…$15.00
DOI: http://dx.doi.org/10.1145/3133956.3133975

1 INTRODUCTION

In Android, the applications are divided into components, which can be conveniently reused by other applications mainly through a message passing mechanism called *intent* in Android Inter-Component Communication (ICC) model. However, ICC model is becoming the main reason for the explosive component hijacking attacks [4, 6, 16, 31, 33, 34, 41], which cause information leakage or even financial loss.

As one type of components, a service can be invoked by using either an explicit intent or an implicit intent. Explicit intents specify the component names or package names explicitly. They are typically used by developers to start components in their own applications (i.e., same origin apps) or the well-known third party services, since the developers already know the component name or package name of the service to be called. In contrast, implicit intents only describe the type of action to perform and allow the system to find a component on the device to perform the action [17]. For brevity, we call the service invocations through implicit intents as *implicit service invocations* and the service invocations through explicit intents as *explicit service invocations*. Researchers found that attackers can launch a service hijacking attack by exploiting the implicit service invocations [10]. By crafting a malicious application that provides a service matching to an implicit intent requested from a victim application, the attacker may make the system choose its malicious service from multiple matching services to serve the victim application under the condition that the malicious service has the highest priority in the service list.

To remove the vulnerabilities introduced by the implicit service invocations, since Android 5.0 (API level 21), Google banned the service components from being invoked through implicit intents [19]. When a service is invoked implicitly, the application will simply crash. By enforcing this forbidden policy, Google expects to resolve the service hijacking attacks by forcing application developers to convert their implicit service invocations to explicit ones.

In this paper, we evaluate the impacts of the implicit service forbidden policy on application developers and its effectiveness on removing vulnerable service invocations by analyzing two sets of the same applications downloaded from Google Play Store before and after the forbidden policy is enforced. Our study focuses on answering two questions. First, how well is the implicit service forbidden policy being accepted by application developers? A statistics analysis [28] shows that 78.1% applications invoking the

service components and in which about 30% services are started through implicit intents. Second, can the forbidden policy resolve the service hijacking attacks in Android 5.0 and higher?

To answer these two questions, we develop an analysis framework called ISA [1] to systematically study the vulnerabilities in service invocations. First, we employ android-apktool [2] as a preprocessor to convert the APK files into smali files, since smali codes do not require disassembling Java code and avoid introducing inaccuracy into the results. Next we develop a static intent analyzer to discover all intents that are consumed for invoking the service components in the smali codes. We develop a reachability verifier to check if the service invocations found by our static intent analyzer are reachable from certain entry points of the apps. We perform a depth first search to traverse the code and check if there is a calling chain from an entry-point to the service invocation APIs. Since we do not need to save the edges and nodes of the full call-graph, the reachability verifier can achieve a good performance. Finally, we develop an vulnerable service invocation analyzer to identify the vulnerable service invocations.

Next we apply our analysis framework on two sets of the same 1390 popular applications downloaded from Google Play Store before and after the implicit service invocations are forbidden. We denote the application dataset downloaded between August 2014 and October 2014 as "Old Apps", since the forbidden policy has not been enforced. We denote the application dataset downloaded in May 2017 as "New Apps", since the forbidden policy has been enforced for 30 months.

Our experimental results show that the forbidden policy effectively reduces the number of vulnerable service invocations from 643 to 112, namely, 82.58% reduction. However, the forbidden policy fails to resolve all service hijacking attacks - 57 among the 112 invocations make 36 applications still vulnerable to hijacking attacks, which can be misused to steal user bank account or VPN login credentials. Since the analyzed applications are the most popular applications from Google Play Store, more than 500,000,000 users could be impacted by these attacks.

After a detailed analysis of the remaining 112 vulnerable service invocations, we discover that the key reasons for the remaining vulnerable invocations are two-fold. The first one is the difficulty in determining the package names for certain third party services, due to the various types of third party services and the difficulty on verifying the trustworthy of the apps. The vulnerable invocation reduction from 643 to 112 is due to the resolve of the same origin services or the well-known Google third party services, which are easy for the developers to determine the package names or for the Android system to identify the suitable applications (e.g., setting higher priority to the same origin and the Google services). However, for the other not well-known third party services, the reduction rate is only 23.46%. We call this type of third party service as "the other third party service", which is the main challenge to resolve the service hijacking attacks.

To avoid explicitly specifying the class or package name for the other third party service, developers are more frequently calling the queryIntentServices() and resolveService () APIs [20] to help convert implicit intents into explicit intents. When calling `queryIntentServices (Intent intent, int flags)` API, the Android system returns a list of services installed on the mobile phone matching the implicit intent in the parameter. The `resolveService (Intent intent, int flags)` API returns the first service in the service list provided by `queryIntentServices ()`. Since the matching rules of `queryIntentServices()` is the same as the vulnerable ones used by `bindService()` and `startService()`, the service invocations through explicit intents converted by `queryIntentServices()` and `resolveService()` may still suffer service hijacking attacks on Android 5.0 and higher. Moreover, we find that attackers may misuse this ranking rules to disable a victim service invocation or crash a victim application. We call the service invocations through explicit intents converted by these two APIs as "the resolved service invocations".

The second reason lies in the difficulty for all the developers to correctly update the applications in time, especially when the services are invoked through outdated SDK or reuse of the outdated sample codes. For example, among the 112 vulnerable service invocations in "New Apps", 62 are residue implicit invocations, and more than half are Google or same origin services. 79.03% residue ones remain implicit since they are invoked through outdated SDKs or reusing outdated sample codes, in which the services are invoked implicitly. There are even some SDKs in which the services are invoked implicitly in the latest versions. After analyzing the Android source codes, we find that when an application's *targetSdkVersion* attribute is set lower than 21 (i.e., API Level of Android 5.0), implicit service invocations are still allowed even on Android 5.0 and higher platforms. 61.29% implicit service invocations in "New Apps" are vulnerable to hijacking attacks rather than app crash.

According to our analysis results, we propose several countermeasures to further mitigate the service hijacking attacks. We suggest an optimization in the ranking rules of the implicit or resolved service invocations, i.e., giving a higher priority to the same origin and the Google third party service. This simple optimization could reduce 44.64% vulnerable service invocations in the "New Apps", among which 72% are vulnerable to hijacking attacks. For the other third party service, we propose a market-based service ranking algorithm to increase the difficulty for attacker to manipulate the ranking of the service list. Several serious unresolved attacks in "New Apps", e.g., stealing the VPN login credentials, could be defeated with this solution. We also discuss two other countermeasures, namely, signature-based verification and SDK hardening, which are promising to mitigate the service hijacking attacks.

2 BACKGROUND

2.1 Service Components and Intents

A *service* is an Android Application component that performs operations in the background without a user interface. Each Service has a corresponding <*service*> declaration in the application *Manifest.xml* file and includes an attribute named *exported* to define if the service can be started in another application. If the *exported* attribute is set to `false`, the service can only be started by components in the same application. In addition, the service may define one or more < *IntentFilter* > to specify the types of intents that the service can respond to. There are two ways to allow a service be

[1] ISA stands for Implicit Service Analysis.

started by other applications. First, its *exported* attribute should be set to `true`. Second, when the *exported* attribute is not set explicitly, at least one Intent Filter is defined.

Service components can be started using either *startService()* or *bindService()* API functions. The *startService()* function performs a single operation with no return values, and the started service runs in the background indefinitely even if the caller component is destroyed. The *bindService()* function provides an enriched communication interface for the caller to interact with the service, but the bound service will be destroyed if all callers have been destroyed.

A messaging object named *Intent* is passed into *startService()* and *bindService()*. There are two types of intents, *explicit intent* and *implicit intent*. Explicit intents specify the component name or package name in the intent explicitly. In contrast, implicit intents only specify some general information, such as the action to be performed, the data to operate on, or the category of the action etc., and delegate the task of evaluating the matching components to the Android system. For a service to be started by an implicit intent, the service should define one or more Intent Filters, which specify the types of intents that the service can respond to. A service matches an implicit intent only when the action, data, and category defined in the intent object match one of the Intent Filters defined by the service. When starting service components through an implicit intent, the Android system calculates the matching values of all the Intent Filters to this intent. If one intent matches multiple services, the system selects the proper service automatically in the background.

2.2 SDK Version

Each application has a corresponding *<uses-sdk>* declaration [22] in its *Manifest.xml* file, which defines the application's compatibility with one or more versions of the Android platform, by means of an API Level integer. In total, three attributes could be defined in the *<uses-sdk>* declaration, i.e., $minSdkVersion$, $maxSdkVersion$, and $targetSdkVersion$. The first two attributes designate the minimum and maximum API Levels designed for the application to run. In other words, installing the application on a system with the API Level lower than $minSdkVersion$ or higher than $maxSdkVersion$ is not allowed. $minSdkVersion$ should be declared in each app, while $maxSdkVersion$ is no longer checked or enforced beyond Android 2.0.1. $targetSdkVersion$ designates the API Level that the application targets at, and its default value is set to the value of $minSdkVersion$. This attribute informs the system that the application has been tested against the target SDK version and the system should not enable any compatibility behaviors to maintain the application's forward-compatibility with the target version. The application is still able to run on older versions down to $minSdkVersion$.

2.3 Service Hijacking Attacks

Originally implicit intents are provided by Android to help ease the developer's working load with more flexibility. For instance, when an application needs some complex services such as image enhancement or object detection, instead of coding our own version of services, we can pass on the operation request and image data to the Android system, which will pick an available application, e.g. OpenCV Manager, that has implemented those services and opens to third-party apps.

Later, researchers found that implicit service invocations are not secure and may suffer from service hijacking attacks [10]. Suppose one application defines a service A that includes at least one Intent Filter and can be called through implicit intent by other apps. An attacker can launch service hijacking attacks by crafting a malicious application that provides a service A* to match the same implicit intent but with higher priority. When a victim application sends the requests for calling service A to the Android system, the system is responsible for automatically selecting the service with the highest priority (i.e., the malicious service A*) from multiple matching services and uses it to serve the victim app.

Android sorts service priority using five Intent Filters' fields, namely, $priority$, $perferredOrder$, $isDefault$, $match$, and $system$, in the decreasing priority order.

- $priority$ is the declared priority of this Intent Filter. The value of $priority$ must be greater than -1000 and less than 1000, higher value means higher priority. The default value is 0.
- $preferredOrder$ represents the user's preference. A higher value means higher priority; however, currently this value is set to the default value 0 for all services and cannot be changed.
- $isDefault$ denotes if an Intent Filter has specified the *Intent.CATEGORY_DEFAULT* attribute, which means a default action can be performed.
- $match$ is the system's evaluation on how well the Intent Filter matches the intent, calculated according to five Intent Filter attributes including $action$, $categories$, $type$, $data$, and $scheme$.
- $system$ defines if it is a service defined in a system application. A system application has a high priority than user applications.

The three fields $priority$, $isDefault$, and $match$ are related to the attributes set in the Intent Filter and thus can be misused by malicious developers to craft a malicious service with higher priority to hijack a victim service. When there are multiple services with the same matching value to an implicit intent, the services will be ranked according to alphabetical order of the package names. Thus, the attacker may misuse the package name to give its service a higher priority. Since Android 5.0, Android suggests to use explicit intents only, and it claims that when a service is invoked through implicit intent, the system simply throws an exception and the application invoking the service implicitly will crash [19].

3 METHODOLOGY

We develop an analysis framework called ISA to systematically analyze the vulnerabilities in service invocations before and after the implicit service invocations are forbidden. As shown in Figure 1, the ISA framework consists of four major components: *Preprocessor*, *Static Intent Analyzer*, *Reachability Verifier*, and *Vulnerable Service Invocation Analyzer*. The preprocessor is responsible for converting APK files into smali files to facilitate further analysis. The static intent analyzer then processes smali files to find out all explicit and implicit intents being consumed for invoking service components.

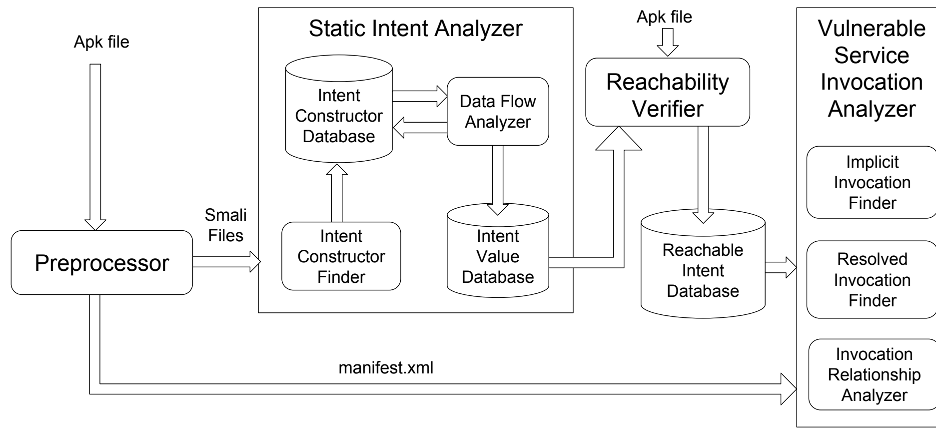

Figure 1: The ISA Analysis Framework

Next, the reachability verifier filters out service invocations in the dead codes and provides a database that only contains reachable service invocations. Finally, the vulnerable service invocation analyzer finds out all the vulnerable service invocations.

3.1 Preprocessor

Android applications are packaged as APK files, which are ZIP compressed file containing compiled bytecode and additional metadata such as manifest.xml file and resource files. To reduce the failures on disassembling the Java code [13, 24], we use `android-apktool` tool [2] to unpack the APK files and obtain the corresponding smali files and manifest.xml file. More importantly, the smali code contains almost all necessary information to facilitate the data flow analysis. For instance, for the override functions with the same function name and different parameters, since the function prototype can be provided when the function is invoked, we can precisely trace the data flow across the function boundary.

3.2 Static Intent Analyzer

We conduct a static data flow analysis on the smali files to discover all explicit and implicit intents that are consumed for invoking the service components. It contains two modules, *Intent Constructor Finder* and *Data Flow Analyzer*. The intent constructor finder is responsible for obtaining a list of methods in which an Intent is constructed and then recording those methods in an *Intent Constructor Database*. The data flow analyzer conducts a static data flow analysis to find all intents that are used to invoke services through `startService()` and `bindService()`, and then records the intent values in an *Intent Value Database*.

3.2.1 Intent Constructor Finder. As the starting point of our analysis, this module finds the list of all methods in which new intents are constructed. It can be done by simply grepping the intent construction code (i.e., *"new-instance v*, Landroid/content/Intent;"*) in the smali codes.

3.2.2 Data Flow Analyzer. The data flow analyzer parses the methods in the intent constructor database and records the intents for invoking services in an intent value database. Figure 2 shows the algorithm of data flow analysis on Intents for each method in the intent constructor database. The process of intents can be divided into two categories: *intra-method intent processing* and *inter-method intent processing*.

```
DataFlowAnalyzer ( SmaliFilePath, Method, IntentList )

Begin
    If ( IntentList Not Null ) TemporaryIntentList.Add(IntentList);

    //Read From the Begin Of the Method Line By Line
    While (( line = ReadNext ( SmaliFilePath:Method ) != End of Method )
    {
        Switch line:
         Case: Invoke Intent Create Functions       TemporaryIntentList.Add;
         Case: Invoke Intent Process Functions      TemporaryIntentList.Update;
         Case: Invoke Intent Consume Functions      Intent Value Database.Add;
         Case: Invoke Auxiliary Functions         Save / Update Related Variables
         Case: Invoke Method1 ( Intent1 in TemporaryIntentList ):
            SmaliFilePath1=GetSmaliFilePath(Method1);
            DataFlowAnalyzer(SmaliFilePath1, Method1,Intent1);
         Case: Return Intent1 in TemporaryIntentList:
            MethodReturnIntentList.Add(Method,Intent1);
            Store Method's caller methods into the Intent Constructor Database
         Case: Put Intent1 in TemporaryIntentList Into A Global Variable GVar
            GlobalValueList.Add(GVar,Intent1);
            Store methods in which GVar is Get into the Intent Constructor Database
         Case: Get GlobalVariable GVar Not In GlobalValueList
            Find MethodList in Which GVar is Put
            Foreach(Method1 in MethodList)
            {
                SmaliFilePath1=GetSmaliFilePath(Method1);
                DataFlowAnalyzer(SmaliFilePath1, Method1,NULL);
            }
         Case: Invoke Method1 in MethodReturnIntentList
            Intent1=MethodReturnIntentList.get(Method1);
            TemporaryIntentList.Add;
         Case: Get GlobalVariable GVar in GlobalValueList
            Var=GlobalValueList.get(GVar);
            if(Var is Intent) TemporaryIntentList.Add;

    }
End
```

Figure 2: The Algorithm of Data Flow Analysis.

Intra-Method Intent Processing. Since we need to obtain the intent attributes such as *action*, *component name*, *package name*, and *class name*, rather than the data passed through the intent such as the URI data, we construct a framework model focusing on Android classes and APIs that handle related intent attributes. For other APIs encountered during the data flow analyzing, we only record the APIs invoked. In total, we identify five Android classes that are critical for the analysis of intent attributes, namely, `android.content.Intent`, `android.content.ComponentName`, `android.content.Context`, `java.lang.Class`, and `java.lang.String`, as shown in Table 1.

Table 1: Intent-related Classes and APIs

Class	API Prototype	Category
android.content.Intent	Intent(Context, Class) Intent(Intent) Intent(String) Intent(String, Uri, Context, Class) Intent(String, Uri) Intent()	Intent Creating Functions
	setClass(Context, Class) setClassName(String, String) setClassName(Context, String) setComponent(ComponentName) setPackage(String) setAction(String) queryIntentServices(Intent,Flag) resolveService()(Intent,Flag)	Intent Processing Functions
android.content.ComponentName	ComponentName(String, String) ComponentName(Context, String) ComponentName(Context, Class) createRelative(String, String) createRelative(Context, String) unflattenFromString(String)	Auxiliary Functions
java.lang.Class	getClassName() getPackageName() getShortClassName() getCanonicalName() getName() getClass() forName(String)	
java.lang.String	Several String Processing Functions	
android.content.Context	getPackageName()	
	bindService(Intent,ServiceConnection,int) startService(Intent)	Intent Consuming Functions

The life time of an intent variable can be divided into three phases, *intent created*, *intent processed* and *intent consumed*. Accordingly, the list of APIs in Table 1 can be divided into intent creating functions, intent processing functions, and intent consuming functions. First, the intent creating functions include all constructor functions of the intent class. When encountering these functions, a new intent will be stored into a temporary list named *TemporaryIntentList*. We ignore all URI variables passed in as parameters. Second, the intent processing functions include configuration functions used to set the attributes of the intent. We focus on the attributes related to our analysis, such as `action`, `class`, `package`, and `component`, etc. When these functions are invoked, the attributes of the corresponding intent in the TemporaryIntentList may be updated. If the intent is processed through `queryIntentServices()` or `resolveService()`, it will be marked in the TemporaryIntentList. Third, the intent consuming functions include the functions that consume an intent to start a service component. When encountering these functions, the corresponding intent in the TemporaryIntentList will be stored into the intent value database along with the invocation position of the intent consuming functions.

In addition, we identify some auxiliary functions that are related to the attributes of intents and list them in Table 1. For instance, the functions in `android.content.ComponentName` class are used to construct a new component name, and the results will be used to set the component name of an intent. As another example, the functions in `java.lang.Class` class are used to get the name of a class, and the results are usually used to set the class name of an intent. Since most attributes of the intent is described by a string, the string processing functions are also considered.

Inter-Method Intent Processing. There are four cases in which an Intent's data flow will come across the method boundary, marked as bold in Figure 2. First, when an intent in the TemporaryIntentList is passed to another method as a parameter, the data flow will go into the new method, which will add the intent in its own TemporaryIntentList. Second, when one method containing an intent in the TemporaryIntentList is returned from another method, we insert the corresponding method name and the return value in a MethodReturnIntentList for each application. In the meantime, all methods invoking this method will be added into the intent constructor database. Figure 3 shows a sample code from an application named "aa.apps.dailyreflections-1". In code snippet (a), an intent (we call it *Intent1* afterwards) is created and returned in one method `zzqS()`. When analyzing the code, Intent1 is inserted into method `zzqS()`'s TemporaryIntentList. In the mean time, <`zzqS()`, Intent1> key value pair is inserted into the app's MethodReturnIntentList. Next we will search the smali codes to find out all the methods invoking `zzqS()` and then insert them into the intent constructor database. As shown in code snippet (b), `zzqS()` is invoked in method `zzcH()`, therefore method `zzcH()` is inserted into the intent constructor database. When analyzing the method `zzcH()`, Intent1 will be inserted into method `zza()`'s TemporaryIntentList, since the return value of method `zzcH()` is passed into the method `zza()`. Finally, in method `zza()` as shown in code snippet (c), Intent1 is consumed by the `bindService()` function to start a service component. Now Intent1 is inserted into the intent value database.

```
(a) Intent is constructed and returned in a method zzqS()

 .method public zzqS()Landroid/content/Intent;
   ...
   // Intent is constructed
   new-instance v0, Landroid/content/Intent;
   iget-object v1, p0, Lcom/google/android/gms/common/internal/zzm\$zza;->zzSU:Ljava/lang/String;
   //Set action of the Intent
   invoke-direct {v0, v1}, Landroid/content/Intent;-><init>(Ljava/lang/String;)V
   const-string v1, "com.google.android.gms"
   //Set package name
   invoke-virtual {v0, v1}, Landroid/content/Intent;->setPackage(Ljava/lang/String;)Landroid/content/Intent;
   move-result-object v0
   ...
   // Return the Intent
   return-object v0

(b) zzqS() is invoked to fetch the Intent in another method zzcH()

 .method public zzcH(Ljava/lang/String;)V
   ...
   //Intent is fetched through calling zzqS()
   invoke-virtual {v2}, Lcom/google/android/gms/common/internal/zzm\$zza;->zzqS()Landroid/content/Intent;
   move-result-object v3
  ...
   //Intent is passed into zza() method
   invoke-virtual/range {v0 .. v5},Lcom/google/android/gms/common/stats/zzb;->zza(Landroid/content/Context;
   Ljava/lang/String;Landroid/content/Intent; Landroid/content/ServiceConnection;I)Z
  ...

(c) In method zza(), the Intent is consumed by the bindService() method
   ...
   //Intent is consumed by the bindService() method
   invoke-virtual {v2, v1, v3, v4},
     Landroid/content/Context;->bindService(Landroid/content/Intent;Landroid/content/ServiceConnection;I)Z
```

Figure 3: Sample Code of Returned Intent.

Third, when one intent in the TemporaryIntentList is stored into a global variable, the key and value for this global variable is stored into a key value mapping list named *GlobalValueList* for each application. In the meantime, all methods consuming this global variable will be added into the intent constructor database. There are several ways to create a global variable in Android application, such as defining a global static variable in the class and using the APIs in SharedPreferences [21] for applications to conveniently store, read, and write a key-values collection. Figure 4 shows a sample code from an application named `com.tvrsoft.santabiblia-13`. In Figure 4 (a), one intent is created in one method and saved into the global variable `Lcom/google/android/gms/auth/GoogleAuthUtil;->Dp:Landroid/content/Intent;`. In another method, the intent is obtained from the global variable and used by the `bindService()` function to start a service component, as shown in Figure 4 (b).

Fourth, when an intent is obtained from a global variable not included in the GlobalValueList, we search and find all methods in which the global variable is put, perform data flow analysis on each method, and set global variable obtained in each method as a potential value of the intent.

(a) Intent is constructed and saved into a global variable in method ppbQ()

```
.method public ppbQ()V
    ......
    // Intent is constructed
    new-instance v0, Landroid/content/Intent;
    invoke-direct {v0}, Landroid/content/Intent;-><init>()V
    const-string v1, "com.google.android.gms"

    // PackageName is set for the Intent
    invoke-virtual {v0, v1}, Landroid/content/Intent;->setPackage(Ljava/lang/String;)Landroid/content/Intent;
    move-result-object v0
    sget-object v1, Lcom/google/android/gms/auth/GoogleAuthUtil;->Dn:Landroid/content/ComponentName;

   // ComponentName is set for the Intent
    invoke-virtual {v0, v1},
Landroid/content/Intent;->setComponent(Landroid/content/ComponentName;)Landroid/content/Intent;
    move-result-object v0

   // Intent is saved into a global variable
Lcom/google/android/gms/auth/GoogleAuthUtil;->Dp:Landroid/content/Intent;
    sput-object v0, Lcom/google/android/gms/auth/GoogleAuthUtil;->Dp:Landroid/content/Intent;
```

(b) In method ssO(), Intent is obtained from the global variable and consumed by the bindService()

```
.method public ssO()V
    ......
   // Intent is fetched from a global variable
    sget-object v3, Lcom/google/android/gms/auth/GoogleAuthUtil;->Dp:Landroid/content/Intent;

    const/4 v4, 0x1

     invoke-static {v1, v3},
    // The fetched intent is consumed by the bindService() method
    invoke-virtual {v1, v3, v2, v4},
Landroid/content/Context;->bindService(Landroid/content/Intent;Landroid/content/ServiceConnection;I)Z
```

Figure 4: Sample Code of Storing Intent in a Global Variable.

In all above four cases, if one attribute of an intent (or the intent variable) depends on certain input parameter of a callee method, we need a backward search for the caller method and start the analysis on the intent from that method again, until we can determine the attribute (or intent) values. If one attribute value of the intent (or the intent variable) is a return value of an abstract method, the smali code only depicts the name of the abstract value. In this case, we need to find all implementation methods and set return values of all methods as the potential values of the attribute (or intent). If one attribute value of the intent (or the intent variable) is a return value of a class inherited from a super class, the smali code only depicts the name of the inheritance class while the implementation is in the super class, we need to find the implementations method in the super class and set the return values of the method as the value of the attribute (or intent).

Figure 2 shows that the data flow analysis is a depth first recursive algorithm. We set two limits to prevent the analysis from entering a dead loop and ensure the algorithm can finish in limited time. First, we empirically set the nesting level to 5 in our implementation, since through manually verifying applications in the ANDROID_WEAR category, 5 is large enough to analyze the intent attributes. Second, when a method is recursively invoked by itself, we will only analyze the method once.

3.3 Reachability Verifier

Since we only concern about the vulnerable service invocations that can be truly triggered in the apps, we develop a reachability verifier to check if the service invocations found by our static intent analyzer are reachable from certain entry points of the apps. All intents related to the reachable service invocations are saved into a *Reachable Intent Database*. First, we need to find all the application entry points; however, different from traditional Java Apps, Android applications are composed of four types of components rather than including an entry point method such as main(), and all the component lifecycle methods and callback methods can serve as entry-points for Android apps. To solve this problem, similar to FlowDroid [4], we generate a dummyMain() for each App, where the dummyMain() includes all lifecycle methods and callback methods.

Then we build the inter-procedure call-graph for the application and traverse all call-graph paths to verify the reachability of the service invocations. If we can find a path from one entry point to the position of invoking startService() or bindService(), the service invocation will be marked as reachable. IC3 [33] provides two algorithms, spark [26, 37] and CHA [11], to construct the inter-procedure call graph on Android apps. However, when we run the two algorithms to build the call graph on 1390 popular applications from Google Play Store, we find that they cannot finish or provide any useful results on 38% applications within 10 hours. Similar results are also reported in Mudflow [5].

Fortunately, our reachability analysis does not need to build a full call-graph. By performing depth first search to traverse the code directly using CHA algorithm, we do not need to save the edges and nodes of call-graph. When there is a calling chain from the entry-point to the service APIs, we consider these APIs as reachable and they can be triggered at certain points. In this way, we can filter out the service invocations in the dead code that will never be triggered.

3.4 Vulnerable Service Invocation Analyzer

The reachable intent database generated by the reachability verifier includes both explicit and implicit intents, so the vulnerable service invocation analyzer consists of three modules to filter out explicit service invocations and provide further analysis on vulnerable service invocations. First, the *Implicit Invocation Finder* finds out the services invoked via implicit intents, i.e., intents without setting

package, class, or component. Second, the *Resolved Invocation Finder* find the service invocation points, which are started through intents processed by queryIntentServices() and resolveService(). Without being processed correctly, these service invocation points may introduce service hijacking attacks or DoS attacks. Detail can be found in Section 4.3. Third, the *Invocation Relationship Analyzer* determines if a service invocation is for same origin service or third-party service by comparing the intent values with the services defined in the manifest.xml. If the class or action attribute of an intent is defined in the manifest.xml, it is a Same Origin invocation. Implicit intents for the same origin services can be easily converted to explicit ones by the application developers; while the third-party services may be correctly converted if those services are well-known, such as services provided by Google.

The vulnerable service invocation analyzer generates three analysis results including (i) a list of implicit service invocation position and the corresponding intent value; (ii) a list of service invocation position and corresponding intent value, where the intent used to start the service is processed by queryIntentServices() and resolveService(); and (iii) the relationship of each service invocation, i.e, if the service invoked is defined in the same application or by a third party application.

4 EVALUATION

In this section, we first depict the application datasets used in our study. Then, we evaluate the effectiveness of removing vulnerable service invocation by directly banning implicit invocations. Finally, we report the unsolved vulnerable service invocations.

4.1 Android Application DataSets

We focus on analyzing the applications in Google Play Store [18]. To study the real impacts of banning implicit service invocations, we download two datasets, one before the implicit service invocation was banned and one after the implicit service invocation had been banned. First, we downloaded the top 100 applications for all the 34 application categories from Google Play store in May 2017, which include 3251 applications in total. After removing duplicated applications in more than one categories, we have 3156 apps.

We meet some challenges to collect applications that were uploaded to Google Play Store before the implicit invocation is forbidden (i.e., November 2014), since Google Play Store only provides the newest-version application downloads. Fortunately, we find an online application archive provided by the PlayDrone Project [1, 39], which includes 1,490,097 Android applications crawled from Google Play Store in year 2014.

Among the 3156 applications downloaded in 2017, we can find 1390 applications in the PlayDrone dataset. Therefore, we obtain two datasets containing the same set of 1390 popular applications with two different versions. We denote the application dataset downloaded from Google Play Store in May 2017 as "New Apps", as they are downloaded after the implicit service invocation has been banned almost 30 months. 99.4% applications in the "New Apps" dataset have been updated after implicit service invocation was forbidden. We name the application dataset downloaded from PlayDrone archive as "Old Apps", as they are downloaded from Google Play Store between August 2014 and October 2014, i.e., less than three months before the implicit service forbidden policy is enforced.

4.2 Effectiveness on Removing Vulnerable Service Invocations

We call the service invocations vulnerable to hijacking attacks or DoS attacks as *Vulnerable Service Invocation* , which includes implicit service invocations and vulnerable resolved service invocations. The former ones are services invoked through implicit intents. The later ones are services invoked via explicit intents converted by calling *queryIntentServices()* or *resolveService ()* APIs, where the developers do not verify the package names when adopting the return values of these two APIs. Details of the vulnerable service invocations in the "New Apps" will be discussed in Section 4.3.

In this section, we show how well the forbidden policy reduces the vulnerable service invocations. In general, the service invocation could be divided into two categories, *the same origin* and *third party*. The former one represents the invocation of service defined in the same app; while the later one describes the invocation of service defined in a different app, including the well-known Google third-party services and various other third-party services.

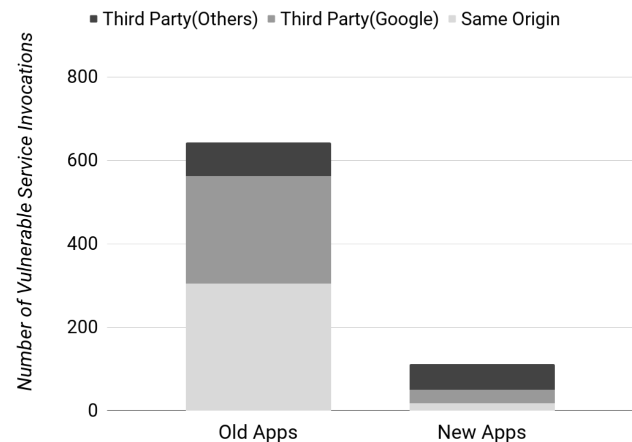

Figure 5: Reduction of Vulnerable Service Invocations

Figure 5 shows that the numbers of vulnerable service invocations in the 1390 applications are 643 and 112, before and after the implicit service invocation is forbidden, respectively. We can see the forbidden policy successfully reduce 82.58% vulnerable service invocations. By carrying out a detailed analysis of the vulnerable invocations, we find that most reduction is achieved by resolving the same origin services (from 304 to 18, 94% reduction) and Google third party services (from 258 to 32, 87.6% reduction). However, the decreasing in the other third party services is very limited, only from 81 to 62 with a 23.46% reduction. Intuitively, it is easy for the developers to convert the same origin implicit service invocation into an explicit one by setting the package name with the return value of *getPackageName()* API. It is also easy for the developers to determine the package names for the Google services, since the number of involved packages is limited and well

known. Actually, among the original 47.28% vulnerable invocations targeting at Google services, only 3 packages are involved, which are Google Play Service ("com.google.android.gms"), Google Play Store ("com.android.vending"), and Google Service Framework ("com.google.android.gsf"). The challenges are to determine the package names for the other third party services, since various packages could be involved in those service invocations. For example, though this type occupies only 12.6% in "Old Apps", it involves 23 different services, and 2 dynamic services whose "action" attributes are extracted from a received message, e.g., a received intent.

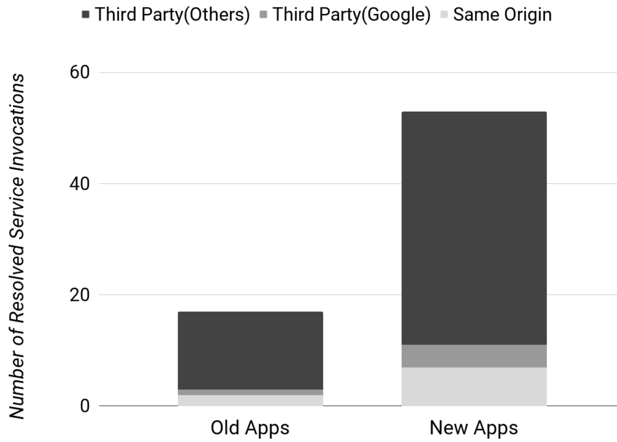

Figure 6: Increment of Resolved Service Invocations.

In addition, we observe that resolved service invocations are used more frequently in the "New Apps", and most of them are used to determine the package name for the other third party services. As shown in Figure 6, though the number of vulnerable service invocations reduces from 643 to 112, the number of resolved service invocations increases from 17 to 53, where 14 (82.35%) and 42 (79.2%) of the resolved service invocations are targeting at the other third party services for "Old Apps" and "New Apps", respectively. When it is difficult to determine the package name for those services, queryIntentServices() and resolveService() are called more frequently to help obtain the matching package name automatically, especially after the implicit service invocation is forbidden. Among the 53 resolved service invocations in "New Apps", 14 are converted from implicit service invocations in "Old Apps", 8 are residue resolved service invocations from "Old Apps". In addition, the number of service types involving resolved service invocations increases from 4 to 17, and the number of involved applications increases from 16 to 31.

4.3 Vulnerable Service Invocations In "New Apps"

In this section, we discuss the remaining vulnerable service invocations in the "New Apps", including implicit service invocations and vulnerable resolved service invocations. In total, 62 implicit service invocations and 50 vulnerable resolved service invocations are involved, and the attacks could be divided into two categories, i.e., service hijacking attacks and denial of service attacks. The implicit service invocation might suffer hijacking attacks when the app's $targetSdkVersion$ is lower than 21, or else it will cause the application to crash (i.e., DoS attacks). And most of the resolved service invocations also suffer attacks, since they are usually used when the package names of the services are difficult to specify. Figure 7 shows that 100% and 94.34% resolved service invocations are vulnerable in the "Old Apps" and "New Apps", respectively. Among the 53 resolved service invocations in the "New Apps", only 3 are not vulnerable, as they verify the package name when adopting the results returned by *queryIntentServices()* and *resolveService ()*.

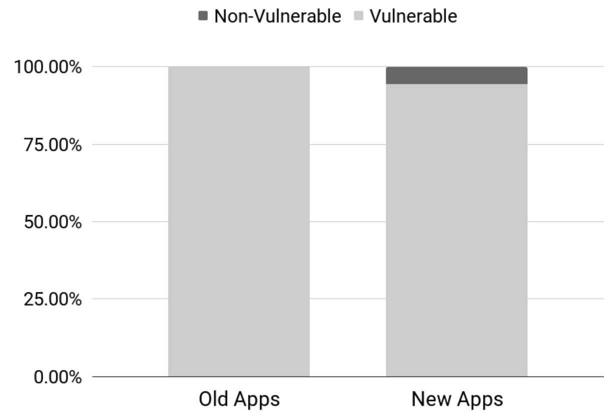

Figure 7: Distribution of the Resolved Service Invocations

Service Hijacking Attacks. Implicit service invocations are vulnerable to hijacking attack when the application's $targetSdkVersion$ is lower than 21. Resolved service invocations are also vulnerable to hijacking attack. Since the ranking rules of *queryIntentServices()*, *bindService()* and *startService()* could be manipulated by the attackers as described in Section 2.3. The statistics data of the "New Apps" shows that it is still easy for an attacker to launch the service hijacking attack. Among the 1390 "New Apps", 722 applications export 2058 services that contain at least one Intent Filter, which may be started through implicit intent. 4.6% applications contain reachable implicit or resolved service invocation codes. Among the 2058 exported services, only 685 services set the *priority* attributes in at least one Intent Filter, and only 72 services set the category to *Intent.CATEGORY_DEFAULT*. Also, among the 685 services, 672 (98.1%) services set the priority value to -500 which is lower than the default value, and only 2 services set to the highest value 1000.

We find 57 service invocations in "New Apps" vulnerable to service hijacking attacks, and 22 service types and 36 applications are involved in these vulnerable service invocations. 38 of the 57 are implicit service invocations with the apps' $targetSdkVersion$ lower than 21, and the remaining 19 are resolved service invocations, in which the developers simply bind/start the first service or all services returned by *queryIntentServices()*. In addition, we find another 31 resolved service invocations that can be hijacked if *queryIntentServices()* API only returns one matching service that is installed by the attacker. When more than one matching services

Table 2: Sampled Service Hijacking Attacks in "New Apps"

AppName	Involved Service	Consequence	Installations	Triggerring Event
jp.nav*	Google In-App Purchase	Sensitive Info Leakage e.g. bank accounts.	500,000,000 - 1000,000,000	App Startup
com.iman*			100,000,000 - 500,000,000	
com.tabta*			1,000,000 - 5,000,000	
com.pico*			500,000 - 1,000,000	
mobi*			1,000,000 - 5,000,000	App Startup (First Time)
com.win*	Samsung In-App Purchase		10,000,000 - 50,000,000	App Startup (On Samsung Platform)
com.cis*	VPN Connecting	Sensitive Info Leakage e.g. VPN Login Credentials.	1,000,000 - 5,000,000	App Startup
com.zen*	VPN Connecting	Same As Above	500,000 - 1,000,000	User Login
com.fox*	Google Messages Transfer	Sensitive Info Leakage	5,000,000 - 10,000,000	App Startup
com.syg*			1,000,000 - 5,000,000	User Login

are returned, the application will either throw an exception or stop starting/binding the service. Table 2 lists several sampled service hijacking attacks in "New Apps". For example, in-app purchasing service is an important service that provides a simple interface for sending in-app billing requests and managing in-app billing transactions. Several popular applications might suffer in-app purchasing service hijacking attacks, so it may cause sensitive information such as bank accounts being leaked. Hijacking the VPN connecting services might cause leakage of sensitive information such as VPN login credentials and data transferred through VPN. As a service used to transfer data between mobile devices and servers, Google Messages Transfer also suffers sensitive information leakage threat. As shown in the column 4 of Table 2, since all these applications are very popular in the Google Play Store, more than 500,000,000 users may be involved. The last column depicts when will these vulnerable invocations be triggered, and 5 of them could be triggered when the applications start up.

In addition to the mentioned attacks in Table 2, all service hijacking attacks will make it easier for the attackers to launch the GUI phishing attacks [9]. Through service hijacking, the attacks could obtain the running state information of an application and could then pop up the phishing user interface accordingly. For example, after an application named "de.affi*" being hijacked, the attacker could pop up a phishing login UI to steal user's login credential.

Denial of Service Attacks. Since current implicit service forbidden policy enhances service security at the expense of sacrificing service availability, it introduces a new type of denial of service attacks. First, all direct implicit service invocations in the applications with $targetSdkVersion$ higher than 21 may cause applications crash. Second, when queryIntentServices() returns a list of more than one services matching to the specific implicit intent, we find that the developers have three choices to process the list which make it vulnerable to DoS attacks. The first one is simply throwing an exception that lead to application crash. The second one is to invoke bindService() or startService() with a `null` intent. The third one is to stop invoking the service. When running on Android 5.0 and higher, the first two choices will crash the app.

In total, we find that 55 service invocations could cause DoS attacks in the "New Apps", where 28 applications are involved. In addition, 53 invocations will cause application crash, and only 2 are choosing service ignorance. Among the 55 invocations, 24 are caused by implicit service invocations, and other 31 are caused by improper processing of the resolved service invocations. If invocation of the service is triggered by entry point functions of an exported component, such as onStart() of an exported Activity component, the DoS attacks could be carried out through manipulating another crafted app. Otherwise, the DoS attacks could only be triggered when users make specific operations themselves in the victim app, such as click a specific user interface.

4.4 Reasons for Implicit Invocation Residue

As described in Section 4.2, one major reason for the residue vulnerable service invocations is the difficulty on determining the package names for the various third party services, which cause more frequent usage of resolved service invocations in "New Apps". However, slow adoption of the forbidden policy is another important reason for the residue vulnerable service invocations. For example, 62 of the 112 vulnerable invocations are residue implicit invoations, which include 28 Google services, 11 the same origin services and 23 the other third party services. 79.03% residue ones remain implicit since they are invoked through outdated SDKs or the reuse of outdated sample codes, in which the services are invoked implicitly. There are some SDKs in which the services are invoked implicitly in the latest versions. In this section, we give a detailed analysis on the implicit invocations to help understand the main reasons for these residue invocations, especially the invocations targeting at the third party services, including Google and the other services, which occupy 82.26% of the total residue implicit service invocations.

We find that outdated SDKs and sample codes are two primary reasons for the implicit invocations of Google services. Table 3 shows the 5 different services involved for the 28 Google service invocations. The first and last columns depict the "Action" attributes corresponding to the services and applications providing the services, respectively. "Implicit Percentage" gives the percentage of the implicit invocations among total ones for each service. "Implicit Reason" shows the causes of the residue implicit service

Table 3: Implicit Invocations of Google Services in "New Apps".

Action	Implicit Percentage	Implicit Reason	App Name
com.google.android.gms.*	6.9%	Outdated SDKs	Google Play Services
com.android.vending.billing.-MarketBillingService.*	50%	Misleading Sample Codes	Google Play Store
com.android.vending.billing.-InAppBillingService.*	1.15%	Outdated Sample Codes	Google Play Store
com.android.vending.licensing.*	54.55%	Outdated SDKs	Google Play Store
com.google.android.c2dm.*	1.17%	Outdated SDKs	Google Services Framework

invocations. For example, the Google Play Services invoked through actions "com.google.android.gms.*" are pervasively used by Android Apps, which contain many important Google services and run as background services in the Android OS. Google provides client libraries to interact with these background services. However, in the client libraries of version 9, 10, and 18-21, some services are invoked through implicit intents. Similarly, implicit service invocation of "com.google.android.c2dm.*" and "com.google.android.vending.licensing" are also caused by outdated libraries. For the two billing services "com.android.vending.billing.MarketBillingService.*" and "com.android.vending.billing.InAppBillingService.*", Google provides sample codes dungeons and Trivial Drive to guide developers on how to interact with them. In the outdated sample codes, implicit intents are used to invoked these two services. The sample codes used to invoke the "com.android.vending.billing.MarketBillingService.*" service are not even updated until May 01, 2017, which could explain the high percentage (50%) for the implicit invocation of this service. In addition, among the 23 implicit invocations for the other third party services, 20 are caused by the outdated SDKs. For example, the most frequent invoked service, "com.bda.controller.IControllerService" is invoked through an SDK, and this service is still invoked implicitly even in the latest SDK "controller-sdk-std-1.3.1".

5 COUNTERMEASURES

Our analysis results show that the one-size-fits-all forbidding solution cannot completely prevent the service hijacking attacks. When the system converts an implicit service invocation to an explicit invocation, the ranking of the service list should not be manipulated by attackers. We first propose an optimization to the ranking rules for the implicit and resolved service invocations, which could block about 90% vulnerable invocations without the attendance of the developers. Then, we propose a market-based service ranking algorithm to increase the difficulty for attacker to manipulate the ranking of the service list. We also discuss two other countermeasures, namely, signature-based service matching and SDK hardening, where the signature-based service matching has been adopted by developers and the SDK hardening may dramatically reduce service hijacking attacks by fixing a small number of popular SDK/libraries.

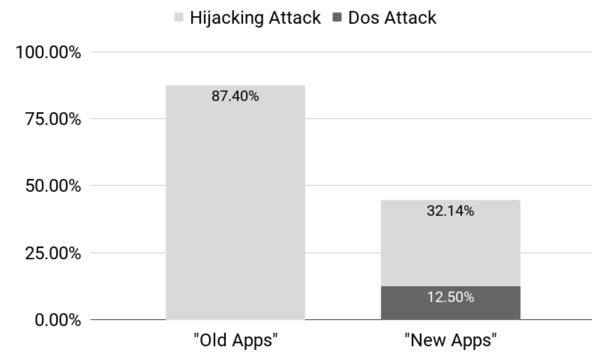

Figure 8: Reduction Percentage with the Optimization

5.1 Optimization in Ranking Rules

As shown in Figure 5, before the forbidden policy is enforced, about 90% vulnerable invocations are targeting at the same origin and Google third party services. Now, the forbidden solution gives the responsibility of converting an implicit invocations to the explicit ones to the developers. However, it is difficult to control the behaviors of millions of developers. An optimization of the ranking rules, i.e., giving higher priority to the same origin and Google services when encountering an implicit or resolved service invocation, could drastically reduce the service hijacking attacks. In addition, it can also mitigate the unresolved attacks in the "New Apps". Figure 8 shows that with the optimization enabled, 87.4% vulnerable service invocations in the original "Old Apps" will be removed. In the "New Apps", the number of vulnerable invocations could also be reduced by 44.64%, among which 72% are vulnerable to hijacking attacks. In addition, 8 attacks in Table 2 could be blocked with this optimization, including the 7 Google service invocations (i.e., 5 In-App Purchase services and 2 Google Messages services) and the VPN Connecting service invocation by the "com.cisco.any*" application (same origin service).

5.2 Market-Based Service Ranking

The basic idea is to delegate the Android App markets such as Google Play to evaluate the trustworthy of each application and then give it a rank based on the market satisfaction. Then, when a service is invoked implicitly, the service with the highest market ranking will be chosen and returned by the system from the service

list. For instance, Google Play Store provides three values to reflect users' satisfaction on one application, namely, *Download Number (DN)*, *Review Score (RS)* and *Review Number (RN)*. Download number reflects the popularity of an application. The higher download number, the higher of the application ranking. The review score and review number reflect the users' satisfaction on the application.

Our market-based service ranking approach calculates a ranking score for each application using Equation 1.

$$Score = \begin{cases} \frac{\omega * DN}{N} + \frac{(1-\omega) * RS * RN}{NR * S}, & \text{if } RS \geq \sigma \\ \frac{\omega * DN}{N} + \frac{(1-\omega) * RS * (N - RN)}{NR * S}, & \text{otherwise} \end{cases} \quad (1)$$

If the review score is higher than a threshold σ, a higher review score and review number will generate a higher ranking score. However, if the review score is lower than the threshold σ, a higher review number means more users are not satisfied on the application, so a higher review number will lower the ranking score. We use ω and 1-ω to weight the download number and the average rating score, respectively. The download number, review score, and review number are normalized by the potential highest download number, the largest review score, and the highest review number of an application, marked as N, S, and NR in the equation. On the Google Play market, N is set to 5,000,000,000, and S is set to 5. NR could be obtained by Google. Our market-based service ranking mechanism is more difficult to be manipulated by the attackers, since the critical values used in calculating the ranking score are difficult to be controlled by attackers. For example, hijacking of the VPN Connecting service invoked by "com.zen*" will be difficult, as the apk ("Cisco Any Connect Client") providing the service is very popular (1,000,000 - 5,000,000 download number) and with high rating score (4.2) and number (7072). Our approach requires Google Play market to maintain a ranking score for each application, and this score should be periodically updated.

5.3 Signature Based Service Verification

Market-based service ranking mechanism works well when the developers have no idea which application or service provider should be trustworthy. If the developer knows which provider of a specific service is trustworthy, a signature-based service verification can be adopted. The developer can invoke queryIntentServices() to get a list of services matching one implicit intent, and then verify the signature of the applications providing the matching services. Only applications provided by the specific providers can be picked to make the intent explicit. This solution provides the flexibility when the developers know which provider is trusted for providing a specific service, since it does not need the developers to upgrade their applications when the package name or class name of the service has changed. Our experiment results show that this solution has already been adopted by some apps. For instance, a service provided by Amazon responding to the action "com.amazon.identity.auth.device.authorization.*" is invoked by 46 apps, and the service invocation method is implemented in an Amazon SDK, which verifies the signature of service provider (i.e., Amazon) before invoking the service.

5.4 SDK Hardening

Since service invocation through an SDK wrapper is popular among Android applications and the majority of remaining implicit service invocations are caused by outdated SDKs, the hardening of SDKs may dramatically reduce the number of implicit service invocations. One way to hardening SDKs is to construct a trusted SDK list for application developers to download the most updated SDK. Alternatively, an incentive mechanism may be developed to motivate the SDK providers for actively updating their SDKs.

6 DISCUSSION

6.1 Accuracy of Static Analysis

There are two challenges that may have impacts on the accuracy of our static analysis, i.e. accurately modeling of the Android framework APIs [31] and accurately analyzing of string variables [27]. The former is caused by the tremendous number and complexity of the APIs and classes in the Android runtime library, and the later is caused by the complexity of the application and string operations. It will introduce huge overhead to accurately process all APIs and strings, and sometime even cause the analysis into dead loop. In this work, we focus on the Intent related framework APIs and string operations. When encountering other framework APIs and string operations, we simply record the value as the invocation of the API. According to our results, we can get precise values for 99.52% of intents, and the average time used to analyze an application is only 50 seconds (including reachability analysis). However, the average time of IC3 is 232 seconds for the 62% applications generate results. One problem introduced by simplifying the modeling is that we may not accurately determine the values for the condition branch. To solve this, our solution records all potential values for the intents in different conditions.

To evaluate the accuracy of our static analysis tool, we manually verify the implicit service invocations in the two datasets, which include 300 and 62 direct implicit service invocations in the "Old Apps" and "New Apps", respectively. Among the 362 invocations, we found that 353 were real implicit service invocations. The 9 false positive implicit invocations could be classified into two categories. First, the SDK version is considered in 7 invocations, and the services will only be invoked implicitly when the SDK version is less than 19 (i.e., Android 4.4) or 21 (i.e., Android 5.0), respectively. In such case, the application still suffers the service hijacking attacks when running on the low version Android system. Second, 2 false positive is due to false classification of the package name setting through the reflection call of the *setPackage()* method, since reflection calls are not considered in our static analysis tool.

6.2 Accuracy of Reachability Analysis

Our static intent analyzer collects all services invocations through finding the `bindService()` or `startService()` calling points in the apps; however, some calling points may reside in dead code that will never run. Thus, we develop the reachability verification to exclude those false positive. Similar to IC3 [33] and FlowDroid [4], our tool cannot deal with the implicit dataflow of reflection and some callbacks, the runtime string values, and the encrypt calculation. Thus, our method may miss some reachable services invocations. On the other side, since the execution logic of some applications are

hard to be triggered [15, 42], we manually verify the reachability of the vulnerable invocations.

7 RELATED WORK

There are explosive researches on Android security, as shown in the related survey [43], which includes not only the malware detection and system protect mechanisms on different Android software stack layers, but also ecosystem based research, such as repackaging detection and prevention [7, 35, 38, 40, 40].

Component Hijacking Attacks. Component hijacking attack has been discussed since 2011 in ComDroid [10]. CHEX [31] and AppSealer [44] use static taint analysis to detect the privacy leakage and privilege escalation caused by vulnerable exported components. FlowDroid [4], Amandroid [41], IccTA [28], DroidSafe [23] provide flow-sensitive static analysis to detect privacy leaks on Android. Later, HornDroid [8] is developed to provide better accuracy and performance by directly working on smali code. Barros et al. [6] develop a tool called Checker, which can resolve the reflection problem and Intent based implicit control flow with the application source codes are available. Alternatively, DroidRA [29] solves the reflection analysis using COAL solver [25]. HARVESTER [36] combines program slicing with code generation and dynamic execution to analyze the runtime values of the sensitive data during malware analysis. TriggerScope [15] and IntelliDroid [42] improve the detection of hidden malicious logic to help detect malicious apps. In this paper, we focus on service component hijacking attacks and give a systematic analysis. Since the Intent values are critical for the component hijacking attacks, Epicc [34] and IC3 [33] are two tools to analyze Intent values. Particularly, as a successor of Epicc, IC3 reduces the intent value analysis into a composite constant propagation problem and solves it with the COAL solver [25]. It can extract the Intent values used by inter-component communication (ICC) APIs. Unfortunately, our experiments show that IC3 fails to generate results for about 38% applications we analyze.

Statistic Analysis of Android Apps. One reason for the explosive growth of Android application is the loose auditing mechanism adopted by Google Play Market. Therefore, whether the developers works in a secure and rigorous approach will have a significant impact on the overall security of Android ecosystem. Statistic Analysis has been adopted on several works to discover the behaviors of the developers. Felt et al. [14] built an automated testing tool named Stowawy and apply it to a set of 940 applications to detects whether Android developers follow least privilege on the permission requests and found that one-third applications are overprivileged. Enck et al. [13] conduct a static analysis on 1,100 popular free applications to discover the common security problems in the applications, and find that many developers failed to securely use Android APIs. Egele et al. [12] study whether the developers use the cryptographic APIs in a secure fashion on 11,748 applications and find that 88% of them make at least one mistake. Viennot et al. [39] perform a large measurement study on 1,100,000 applications crawled from the Google Play application store to uncover the characterization of the application content such as its evolution over time, library usage etc. Lindorfer et al.[30] analyze 1,000,000 applications to discover the trends in malware behaviors observed from applications. Afonso et al. [3] perform an analysis of the native code usage in 1.2 million Android applications and propose an automatic sandboxing policy for protecting native code. McDonnell et al. [32] conduct a study to find the catching up rate of developers when APIs evolve and find that about 28% of API references are outdated with a median lagging time of 16 months. In this paper, we conduct a study with a different goal, which is to find out the adoption trend of the one-size-fits-all implicit service forbidden solution among the application developers and find that even the implicit service invocations had been forbidden for more than two years, there are 64 applications still vulnerable to service hijacking or DoS attacks.

8 CONCLUSIONS

Because of service hijacking attacks, the implicit service invocations have been forbidden since Android 5.0. In this paper, we revisit the service invocations and evaluate the impacts and effectiveness of disabling implicit invocations by simply throwing an exception. Our experiments show that after implicit service invocations had been banned for 30 months, 36 popular applications still contain codes vulnerable to service hijacking attacks. Moreover, we find that this one-size-fits-all solution not only still suffers from service hijacking attacks, but also introduces a new Denial of Service attack on Android apps, and 28 applications are involved. Finally, we propose a new ranking algorithm on selecting services invoked through implicit intents to mitigate the remaining service hijacking attacks.

ACKNOWLEDGMENTS

We greatly appreciate the insightful comments and constructive feedback from the anonymous reviewers. We would like to thank Xianchen Meng for his work at the early stage of this project. This work is partially supported by U.S. Office of Naval Research under Grant N00014-16-1-3214 and N00014-16-1-3216, the National Key Research and Development Program of China under Grant 2016YFB0800102, the Natural Science Foundation of China under Grants 61572278 and 61472165, Guangzhou Key Laboratory of Data Security and Privacy Preserving, and Guangdong Provincial Key Laboratory of Data Security and Privacy Preserving. The corresponding authors are Yuewu Wang and Qi Li.

REFERENCES

[1] 2014. PlayDrone Android Apps. https://archive.org/details/android_apps. (2014).
[2] 2017. Apktool:A tool for reverse engineering Android apk files. https://ibotpeaches.github.io/Apktool/. (2017).
[3] Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupé, Mario Polino, Paulo de Geus, Christopher Kruegel, and Giovanni Vigna. 2016. Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*.
[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM SIGPLAN Notices* 49, 6 (2014), 259–269.
[5] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 426–436.
[6] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Michael D Ernst, et al. 2015. Static Analysis of Implicit Control Flow: Resolving Java Reflection and Android Intents (T). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 669–679.

[7] Sven Bugiel, Stephen Heuser, and Ahmad-Reza Sadeghi. 2013. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. 131–146.

[8] Stefano Calzavara, Ilya Grishchenko, and Matteo Maffei. 2016. Horndroid: Practical and sound static analysis of android applications by smt solving. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*. IEEE, 47–62.

[9] Qi Alfred Chen, Zhiyun Qian, and Zhuoqing Morley Mao. 2014. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks.. In *USENIX Security*, Vol. 14. 1037–1052.

[10] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing inter-application communication in Android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM, 239–252.

[11] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*. Springer, 77–101.

[12] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 73–84.

[13] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. 2011. A Study of Android Application Security.. In *USENIX security symposium*, Vol. 2. 2.

[14] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 627–638.

[15] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2016. Triggerscope: Towards detecting logic bombs in android applications. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 377–396.

[16] Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2013. Highly precise taint analysis for Android applications. *EC SPRIDE, TU Darmstadt, Tech. Rep* (2013).

[17] Google. 2017. Android Application Fundamentals. https://developer.android.com/guide/components/fundamentals.html. (2017).

[18] Google. 2017. GooglePlay. https://play.google.com/store/apps?hl=en. (2017).

[19] Google. 2017. Intents and Intent Filters. https://developer.android.com/guide/components/intents-filters.html. (2017).

[20] Google. 2017. Service Component. https://developer.android.com/guide/topics/manifest/service-element.html. (2017).

[21] Google. 2017. Shared-Preferences:Saving Key-Value Sets. (2017).

[22] Google. 2017. Uses SDK Element in Android Application Manifest File. https://developer.android.com/guide/topics/manifest/uses-sdk-element.html. (2017).

[23] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe.. In *NDSS*. Citeseer.

[24] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. 2013. Slicing droids: program slicing for smali code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM, 1844–1851.

[25] PSU SIIS Lab. 2014. coal solver. http://siis.cse.psu.edu/coal/. (2014).

[26] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction*. Springer, 153–169.

[27] Ding Li, Yingjun Lyu, Mian Wan, and William GJ Halfond. 2015. String analysis for Java and Android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 661–672.

[28] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 280–291.

[29] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. 2016. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 318–329.

[30] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. 2014. Andrubis–1,000,000 apps later: A view on current Android malware behaviors. In *2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*. IEEE, 3–17.

[31] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 229–240.

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

[33] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 77–88.

[34] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. 543–558.

[35] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2014. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications.. In *NDSS*, Vol. 14. 23–26.

[36] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting runtime values in android applications that feature anti-analysis techniques. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*.

[37] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 49–61.

[38] Stephen Smalley and Robert Craig. 2013. Security Enhanced (SE) Android: Bringing Flexible MAC to Android.. In *NDSS*, Vol. 310. 20–38.

[39] Nicolas Viennot, Edward Garcia, and Jason Nieh. 2014. A measurement study of google play. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 42. ACM, 221–233.

[40] Ruowen Wang, William Enck, Douglas Reeves, Xinwen Zhang, Peng Ning, Dingbang Xu, Wu Zhou, and Ahmed M Azab. 2015. EASEAndroid: automatic policy analysis and refinement for security enhanced android via large-scale semi-supervised learning. In *24th USENIX Security Symposium (USENIX Security 15)*. 351–366.

[41] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. 2014. 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*. ACM, 1329–1341.

[42] Michelle Y Wong and David Lie. 2016. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*.

[43] Meng Xu, Chengyu Song, Yang Ji, Ming-Wei Shih, Kangjie Lu, Cong Zheng, Ruian Duan, Yeongjin Jang, Byoungyoung Lee, Chenxiong Qian, et al. 2016. Toward Engineering a Secure Android Ecosystem: A Survey of Existing Techniques. *ACM Computing Surveys (CSUR)* 49, 2 (2016), 38.

[44] Mu Zhang and Heng Yin. 2014. AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications.. In *NDSS*.