Borrowing Your Enemy's Arrows: the Case of Code Reuse in Android via Direct Inter-app Code Invocation

Anonymous Author(s)

ABSTRACT

The Android ecosystem offers different facilities to enable communication among app components and across apps to ensure that rich services can be composed through functionality reuse. At the heart of this system is the Inter-component communication (ICC) scheme, which has been largely studied in the literature. Less known in the community is another powerful mechanism that allows for *direct inter-app code invocation* which opens up for different reuse scenarios, both legitimate or malicious. This paper exposes the general workflow for this mechanism, which beyond ICCs, enables app developers to access and invoke functionalities (either entire Java classes, methods or object fields) implemented in other apps using official Android APIs. We experimentally showcase how this reuse mechanism can be leveraged to "*plagiarize*" supposedly-protected functionalities. Typically, we were able to leverage this mechanism to bypass security guards that a popular video broadcaster has placed for preventing access to its video database from outside its provided app. We further contribute with a static analysis toolkit, named DICIDer, for detecting direct inter-app code invocations in apps. An empirical analysis of the usage prevalence of this reuse mechanism is then conducted. Finally, we discuss the usage contexts as well as the implications of this studied reuse mechanism.

1 INTRODUCTION

Code reuse, a.k.a. *software reuse*, is a form of knowledge reuse in software development that is fundamental to accelerate innovation. Its practice in software engineering is as old as programming itself [1], and has been exacerbated recently within mobile programming frameworks to respond to the needs for keeping up with market requirements of up-to-date functionalities. The facilities offered by Android in this respect have even enabled a large number of software authors to contribute to the application ecosystem, often with little professional training [2].

Reusability is at the core of the Android ecosystem, which builds on the popular Linux kernel and the Java and XML languages to benefit of the extent of device drivers and software libraries to bootstrap functionality development. Unfortunately the staged compilation process as well as the packaging model makes Android apps straightforward to reverse engineer and copy. This has led to a situation where *cloning* (a.k.a., *repackaging*) is commonplace [3–5]. At an inner-level, Android intents and intent-filters facilitate decoupling and assembling of app components, providing opportunities for reuse of existing components to interact with new components. For example, malware writers are extensively exploring these reuse facilities to *piggyback* malicious code on legitimate app by leveraging events (e.g., SMS incoming broadcast) to trigger malware execution. More generally, *component hijacking* in Android has been largely investigated in the literature [6, 7]: by evading permission checks, an Android app may access resources that it is not allowed to. In this respect, Inter-Component Communication (ICC) analyses [8–13] have been proposed to track data leaks as well as to detect permission redelegations attacks [14]. Further investigations were performed in the literature towards uncovering potential *app collusion* [15, 16], i.e., cases where a set of apps are able to carry out a threat in a collaborative fashion. App collusion is indeed generally associated to information leakage and inter-app communication where developers leverage Android implicit and explicit messaging services to orchestrate legitimate rich scenarios or devise sophisticated attacks.

In this paper, we dissect a less-advertised reuse mechanism that is available in the Android framework, through which developers can invoke a given functionality code implemented in another app. We refer to it as **Direct Inter-app Code Invocation** (DICI). To the best of our knowledge, this mechanism was never mentioned in the Android research literature. DICI is often used in legitimate contexts such as across Google Mobile Services[1] to enable functionality reuse among apps. Nevertheless, as we will demonstrate in Section 2, DICI can be used maliciously to plagiarize other-wise protected functionalities and by-pass standalone app analysis.

The DICI mechanism achieves inter-app interactions without leveraging the Android standard inter-component communication primitives. This mechanism builds on Java reflection and a set of dedicated API methods that are provided within the Android framework. DICI differs from existing reuse mechanisms in various ways: (1) In contrast to cloning, DICI does not actually change nor even copy the reused code inside the attacking app. (2) unlike ICC, DICI can allow the invocation of functionality that is not implemented within an Android component, such as a library function in another app. In other words, DICI widens the reuse surface: with DICI any code can be invoked, not only code that is in specific components such as with ICC. (3) Finally, DICI can be leveraged to implement stealthy code reuse. Indeed, while with ICC the developer of the reuse target may be aware that her code could be reused, it is not necessarily the case for DICI. In Android, a component, such as an Activity or a Service, has its "`exported`" attribute set as "`True`" when the developer wishes to allow ICC from another app. Such a developer may then take appropriate measures to ensure component security. In the case of DICI, a developer of an app is not aware that her code will be invoked by a third party.

The main contributions of our work are:

- We expose a little-advertised reuse mechanism within the Android ecosystem. In particular we demonstrate how it can be leveraged to perform stealthy functionality plagiarism that may not be covered by standard licensing scheme.

ESEC/FSE 2020, 8 - 13 November, 2020, Sacramento, California, United States
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] GMS are the apps by Google that often come pre-installed on Android devices. They are not part of the Android Open Source project

- We develop a static analysis tool, DICIDer, for the detection of DICIs in Android apps. We perform an empirical analysis on the prevalence of DICIs among a large dataset of apps retrieved from the AndroZoo repository [17]. We further provide extensive discussions on how and why developers use DICIs through an analysis of sample cases.
- We propose an example of countermeasure that could be used by developers to protect their apps against DICI.

2 DISSECTION OF THE DICI MECHANISM

We provide a problem statement for the direct inter-app code invocation mechanism (Section 2.1) and showcase some motivating examples of reuse based on this mechanism (Section 2.2).

2.1 Direct Inter-app Code Invocation in Android

Given the lack of related information on the mechanism of Direct Inter-app Code Invocation within the Android research literature, we contribute to the body of knowledge by presented an overview of the mechanism. DICI is a mechanism for inter-app communication (i.e., the possibility for one app to leverage ressources, either functionality or information, from another app during its execution). Figure 3 summarizes how inter-app communication works in Android by illustrating DICI in comparison with the standard ICC (i.e., inter-component communication).

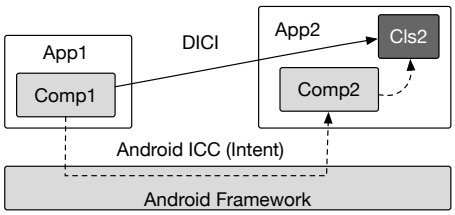


Figure 1: The two types of inter-app communication: Android ICC and DICI.

ICC is the standard mechanism used by developers (as recommended in the Android documentation) to achieve inter-app communication. Its capabilities and challenges have been (and still remain) intensively studied in the research literature. ICC is an Android-specific mechanism that was implemented to enable interaction among Android components, i.e., the basic units that are composed to form apps. Indeed, the four types of components, namely Activity, Service, Broadcast Receiver, and Content Provider, which are responsible for different tasks, cannot directly invoke each other's functionalities. Developers must then rely on specific ICC methods, such as *startActivity()*, to achieve this interaction. As illustrated in Figure 3, an Android ICC is triggered by the Android framework. Thus, there is no direct connection between the source and target components *at the code implementation level*. When the target component of an ICC belong to a different app than the source component, an inter-app communication is implemented.

Beyond ICC, we have come across cases in the practice of Android app development where inter-app communication can be achieved through direct code invocation. We refer to such a mechanism as DICI. As illustrated in Figure 3, DICI can not only (1) bypass the Android ICC mechanism to realize inter-app communication, but also (2) directly invoke non-component code (e.g., standard Java class *Cls2* in Figure 3). The latter capability is not available through the recommended ICC mechanism.

Problem statement. To date, ICC-based inter-app communication has been widely investigated by the research community [10, 16, 18–21]. The literature provides extensive results on tracking information flow through ICC, statically flagging how such inter-app communications are leveraged by attackers to achieve malicious behaviours (e.g., privacy leaks). Studies of how malware is written in bulk through reusing legitimate apps (i.e., piggybacking [22]) have mainly focused on investigating how ICC is relied upon to trigger malicious payload.

Our hypothesis is that DICI, yielding a larger reuse surface, poses challenges that are at least as acute as for ICC. Indeed, DICI may be used by attackers to achieve malicious code invocations while bypassing the security analysis tools which have been focused on ICC-based scenarios. Furthermore, our work raises awareness among the developer community on the possibility of having their functionalities reused without their knowledge by either plagiarists or malicious attackers. Finally, for the research community, exposure to this reuse mechanism in the Android realm re-opens a variety of research directions.

2.2 DICI in action

We highlight the possibilities that offer the DICI mechanism with two motivational use cases. Both cases involve the development of apps that reuse code available in other apps.

(1) `StealthApp` is designed to orchestrate an app collusion scenario for private data leak. In this example, we focus on the possibility of leveraging DICI to hide malicious code in order to increase the chances of escaping detections that are attempted via static analyses.
(2) `TikTokDownloader` showcases the critical possibilities that DICI provides in terms of plagiarism. In this example, functionalities (including backend infrastructure) of one of the most popular apps in the world, namely TikTok [23], are reused to build at no-cost a video-sharing app. In particular, we show that (1) we can reuse several TikTok functionalities, (2) we can provide additional functionalities that are initially forbidden by TikTok, (3) we can even leverage the full infrastructure of TikTok.

2.2.1 Malicious Code Hiding. The International Mobile Equipment Identity (*IMEI*) is a number used as a standard to identify mobile phones. It is considered to be a key private information and should be kept private [24–26]. Consequently, APIs to obtain the IMEI are classically considered in the list of "sources" for data-flow analyses [27, 28], thus facilitating detection of leaks, even when ICC are used. We propose to leverage DICI to orchestrate the leakage of the *IMEI* via SMS: the goal is to build exclusively on code that is implemented from other apps (1) to retrieve then (2) to leak the IMEI. We consider this use case to be reasonable since on a given device it is highly likely to identify another apps that implement a code fragment for sending SMS and another app that has code where the IMEI is retrieved. By doing so, we ensure that there is no explicit code in our developed app (i.e., `StealthApp`) where neither IMEI collection can be matched (e.g., via tracking calls to API) nor

leaking via SMS can be identified. Therefore our implementation of such a collusion, with DICI, challenges the detection of security leaks in StealthApp.

Listing 1 provides an excerpt of the code used in StealthApp to invoke a method (*getDeviceID* at line 13) from another app (org.communicorpbulgaria.bgradio at line 4). Note that in this code the actual API provided by the framework (i.e., android.telephony.TelephonyManager) is hidden. DICI is implemented in this case through reflection after obtaining the context of the app that implements the code to reuse (lines 3-7). Based on this context, the class loader of the app can be obtained (line 8) and used to load relevant classes in the app (line 9-11). The method object is acquired with the class object containing it (line 12-15). Since *getDeviceID*, which is implemented by the target app, is a static method, it is invoked directly to finally get the *IMEI* number (lines 16).

```
1  private String getImei() {
2      String imei = null;
3      Context invokee = this.createPackageContext(
4          "org.communicorpbulgaria.bgradio",
5          Context.CONTEXT_INCLUDE_CODE |
6          Context.CONTEXT_IGNORE_SECURITY
7      );
8      ClassLoader loader = invokee.getClassLoader();
9      Class util = loader.loadClass(
10         "org.ccb.radioapp.components.Utils"
11     );
12     Method getDeviceId = util.getDeclaredMethod(
13         "getDeviceID", Context.class);
14     imei = (String) getDeviceId.invoke(null, this);
15     return imei;
16 }
```

Listing 1: Retrieval of IMEI through reflection for third-party code reuse

Similarly, a method from another third-party app is invoked to send the obtained *IMEI* via SMS as shown[2] in Listing 2. This method is named *sendSms* (line 12), but it cannot be confused with framework APIs for sending APIs. Instead, this method is contained in class CommonUtils (line 9) from app com.globalcanofworms.android.simpleweatheralert (line 3).

```
1  private void sendMsg(String num, String msg) {
2      Context invokee = this.createPackageContext(
3          "com.globalcanofworms.android.simpleweatheralert",
4          Context.CONTEXT_INCLUDE_CODE |
5          Context.CONTEXT_IGNORE_SECURITY
6      );
7      ClassLoader loader = invokee.getClassLoader();
8      Class util = loader.loadClass(
9
         "com.globalcanofworms.android.coreweatheralert.CommonUtils"
10     );
11     Method sendSms = util.getDeclaredMethod(
12         "sendSms", Context.class, String.class, String.class);
13     sendSms.invoke(null, this, num, msg);
14 }
```

Listing 2: Data transfer via SMS through reflection for third-party code reuse

StealthApp performs a malicious behavior, through app collusion, without explicitly implementing any malicious code. As long as all the apps targeted for reuse are available on the users device, its *IMEI* can be leaked and yet, both dynamic and static scanning techniques will systematically fail to spot this leak if apps are analysed individually. In any case, even when the apps are available, it is important to note that the use of reflection makes the use static analysis techniques challenging. While some techniques (e.g., code instrumentation of reflective calls into direct calls with DroidRA [29]) have been proposed in the literature to overcome limitations raised by reflection, these techniques generally target in-app code (e.g., dynamically loaded classes from an extra dex file) Such method would not work for DICI since the method that should be called is not present in the analyzed app.

⚠ **Limitations of the StealthApp use case:** We have developed a naive app colusion system with StealthApp as a proof-of-concept of hiding malicious code with DICI. The goal, with this use case, is not to implement a sophisticated attack. Besides its simplicity, this use case presents several limitations:

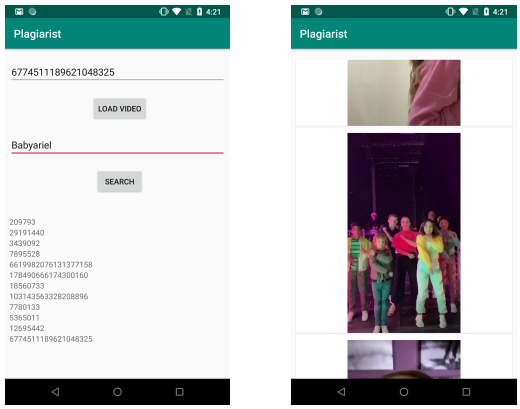
- *Availability of target apps.* The implementation of the malicious behavior depends on the installation status of other apps to orchestrate the app collusion. Their probability of availability on the device could lower the possibility of the execution of the malicious code. Nevertheless, we can expect attackers to leverage the diversity of apps that are shipped with new devices. For example, hackers could list all the functionalities offered by the apps that are already installed on all devices from a specific manufacture, or consider only focusing on popular apps to increase the probability of being able to realise the scenario on millions of devices. Finally, note that the official API PackageManager.queryIntentActivities with *Intent* category set to CATEGORY_LAUNCHER can be used to retrieve at runtime the relevant information on installed apps on the current device.
- *Permissions.* Another limitation is that permissions of other apps will not be granted to StealthApp when StealthApp is invoking their code. Therefore, when a method is protected by a permission, this permission must be granted as well to the app before invoking the third-party code. For our *IMEI* leakage example, the *READ_PHONE_STATE* and *SEND_SMS* permissions are required in StealthApp. Neverthemess, because of the recurrence of permission over-privilege (i.e., apps ask for more permissions than they need) in the Android ecosystem [30, 31], attacks such as the one perpetrated by StealthApp can go unnoticed.
- *Process access.* Finally, it is noteworthy that in the case of ICC, when an app A is "calling" a component of an app B, that component is launched in the process of B, i.e., the target code that is run in B can access the internal data of B. With DICI, when an app A invokes code from B, this code is launched in the process of A, meaning that this code cannot actually access internal data of B. Nevertheless, despite this limitation for accessing more ressources, accessing functionality implementation poses different threats as we will show in the second use case.

2.2.2 Functionality Plagiarism. *TikTok* is a highly popular video-sharing app. It has more than 500 million installs on *Google Play* alone. In line with the necessity to control copyrights of video submitters as well as due to commercial needs to strongly bind users, all the shared videos can only be viewed and downloaded through the single TikTok app. Among other constraints, TikTok does not allow allow batch downloading (i.e., the possibility to download all of the videos of a single user at once). In order to block download requests originating from third-party interfaces, each request need to be appended with a one-time "signature" for the Tiktok server to verify the legitimacy of the request. This "signature" is calculated by an algorithm implemented within the user app with certain information such as user ID, time stamps, etc. These mechanisms are rather effective against the typical cloning (i.e., repackaging

[2]The aforementioned code snippets are simplified with absence of exception handling.

attack) or the reverse engineering of the TikTok app in order to exploit the backend infrastructure and ressources of Tiktok, notably the database of videos.

We will show now that, with DICI, it is actually possible to *reuse* the code of TikTok to achieve the objective of exploiting the TikTok infrastructure. Typically, we were able to implement our own batch downloader, that we call `TikTokDownloader`, to download *TikTok* videos by accessing and plagiarizing the signing algorithm implementation in the *TikTok* app. As shown in Figure 2, the developed app will require just to input a user ID to specify the videos of which user must be downloaded.



(a) Search User ID **(b) Downloaded Videos**

Figure 2: Batch Downloader Snapshots

TikTok implements video search and download via REST endpoints (i.e., an URL where requests can be specified for actions or resoources). However, as endpoints are accessed via requests that are transmitted in plain text with logical structures, they can be obtained, manipulated and used easily by third parties. Thus, to reserve the exclusive use of these endpoints to *TikTok* itself, a one-time "signature" is required to be appended to each endpoint when requesting the server. After investigating the DEX[3] code of TikTok, we identified a method whose obfuscated name is "*a*" within class `com.ss.android.ugc.aweme.app.a.c`, which computes the signature for the app to access the TikTop server ressources. Listing 3 presents the implementation code of method "a" which is the target of our plagiarism scenario.

```
1  private String a(String str) {
2      int i;
3      String userInfo;
4      String str3;
5
6      int serverTime = NetworkUtils.getServerTime();
7      if (serverTime < 0) {
8          i = 0;
9      } else {
10         i = serverTime;
11     }
12     String str4 = str + "&ts=" + i;
13
14     HashMap hashMap = new HashMap();
15     d.a(hashMap, true);
16     String[] strArr = new String[(hashMap.size() * 2)];
17     int i2 = 0;
18     for (String str5 : hashMap.keySet()) {
19         String str6 = (String) hashMap.get(str5);
20         if (str5 == null) {
21             str5 = "";
22         }
23         if (str6 == null) {
24             str6 = "";
25         }
26         int i3 = i2 + 1;
27         strArr[i2] = str5;
28         strArr[i3] = str6;
29         i2 = i3 + 1;
30     }
31
32     userInfo = UserInfo.getUserInfo(i,
         URLDecoder.decode(str4), strArr, "");
33
34     int length = userInfo.length();
35     String substring = userInfo.substring(0, length >> 1);
36     str3 = (str4 + "&as=" + substring + "&cp=" +
37         userInfo.substring(length >> 1, length)) + "&mas=" +
38         com.ss.android.common.applog.i.byteArrayToHexStr(
39             com.ss.sys.ces.a.e(substring.getBytes()));
40     return str3;
41 }
```

Listing 3: Simplified Signing Method from *TikTok*

`TikTokDownloader` implements the DICI mechanism to invoke the method illustrated in Listing 3 in order to sign the endpoints and then request the server with the signed endpoints. It is worth to mention that all of the relevant classes, such as *NetworkUtils* in line 6, will be automatically loaded as well. This constitutes a powerful capability of the DICI mechanism since even native libraries (generally preserved from reverse-engineering due to their machine binary format) can also be reached: for example, in the sample code, *getUserInfo* (line 32), *byteArrayToHexStr* (line 38) and *e* (line 39) are all sensitive code that are embedded in native code.

The usage scenario of our `TikTokDownloader` app is that it is installed on a device where the user already has an account on TikTok. The DICI mechanism in this case has led to the implementation of copyright infringement attacks (since video uploaders did not provide any rights to `TikTokDownloader` to access their content). Another critical point is that DICI allowed to easily plagiarized the TikTok code in a stealthy: `TikTokDownloader` did not copy the code, nor did it rewrite in some way; instead it just invokes it at runtime, a case that is not comprehensively studied in the literature of code plagiarism.

Apps availability and responsible disclosure: We provide on GitHub the source code of both use-case apps as artefacts for further research: *http://shorturl.at/cgkr9*. Both apps have been tested on a *Nexus* 5 device running Android version 8.1.0. We have also responsibly informed TikTok owner company about the risk posed by DICI with respect to the possibility to bypass their security infrastructure to access users copyrighted videos.

3 TOOL DESIGN

Aiming at automatically inferring the usage of DICIs in Android apps, we design and implement a prototype tool called DICIDer, which takes as input an Android APK file and outputs a list of DICI paths that trace how direct inter-app code invocations are planned in the analyzed app. An overview of the working process of DICIDer is presented in Figure 3. Overall, DICIDer follows four steps to pinpoint DICI instances. We now briefly introduce these steps.

3.1 Step (1): Call-graph Construction

DICIs are implemented following a sequence of API calls (e.g., to obtain the third party app context, load the relevant class, invoke

[3] *JADX* is used here for the decompiling, it can be found at https://github.com/skylot/jadx

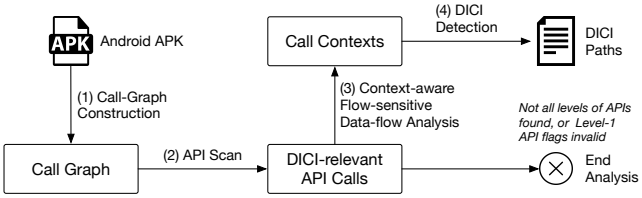


Figure 3: DICIDER static analysis for uncovering DICIs.

the target code, etc.). We thus propose to construct the call graph of the input Android app to facilitate further analyses. To that end, DICIDER relies on the Soot Framework [32] as well as the FlowDroid [27] precise taint analysis tool. To realise this analysis, the apk is first disassembled and the app Dalvik bytecode is transformed into Jimple, the intermediate representation that is leveraged in Soot. Then, this Jimple code is analysed by Soot to yield a call graph of the app.

We recall that Android apps do not come with a single entry-point (e.g., *main* in classical Java applications) to start app execution. Instead, the app can be started from different entry points, from any app components, which complexifies the construction of a single call graph. To address this problem, FlowDroid constructs a dummy main method for analysis. This dummy main method takes into account all the possible entry-points of the app (i.e., components) and their lifecycle methods (e.g., *onStart()*, *onStop()*) as well as all the leveraged callback methods (*onClick()*). The reason why lifecycle methods and callback methods are needed to be explicitly included is that these methods are not explicitly connected at the code level. The prepared dummy main method then enables the Soot to construct the call graph of the app and subsequently to traverse all the app code in their possible execution contexts. Note that Soot implements several in-house call graphs construction algorithms such as CHA and SPARK. While the CHA algorithm is faster than SPARK, it is rather more imprecise [32]. Given that in our work, precision of call graph is a key property to ensure that DICIDER yields good performance, we choose to leverage the SPARK algorithm to build the call graph (with the correct API calling sequences modeled).

3.2 Step (2): API Scan

Once the call graph is constructed, the second step that is unfolded is to identify the relevant APIs that contribute to the realisation of DICIs. Then, one must assess the parameters of these API calls to further confirm potential code reuse scenarios.

API presence detection: DICIDER performs a quick scan over the call graph to check if DICI-relevant APIs of the Android framework are leveraged by the app. The presence of such APIs is a primary condition for the presence of DICIs in the analyzed app. If such APIs do not exist, there is no need to proceed further, and the analysis of the app is safely halted.

Which are the DICI-relevant APIs? In Section 2.2, our use-case description highlighted a sample sequence call of specific Android APIs. Following up on this example, we have carefully investigated APIs that are used in the same principles, and tag them as DICI-relevant.Table 1 enumerates all DICI-relevant APIs considered by DICIDER, along with their implementation class, signature, return type and a textual description. Since DICIs are performed through a sequence of calls of several DICI-relevant APIs, each API may be necessary at different position/level within the sequence. We indicate for each DICI-relevant API the **level** of that API, which represents the position of its call within an instance of DICI call sequence. Generally, a successful DICI needs to involve at least one API in each of the five levels' APIs.

- **Level 1:** Obtain the *context* of another app.
- **Level 2:** Obtain the corresponding *class loader* using the obtained *context* of the other app.
- **Level 3:** Load the *class* (to be directly invoked) of the other app through the obtained class loader.
- **Level 4:** Locate the *constructor*, *method*, or *field* (to be directly reused) from the loaded class.
- **Level 5:** Finally, access the previously located constructor, method, or field reflectively. If the method or field is not declared as *static*, an additional step is needed to instantiate an object of the class.

DICIDER uses the list of DICI-relevant APIs to check whether the analyzed apk contains such APIs. In particular, if the analyzed apk does not contain at least one DICI-relevant API of each of the five levels enumerated previously, the API call sequence is ignored at this stage and DICIDER terminates with no DICI paths detected.

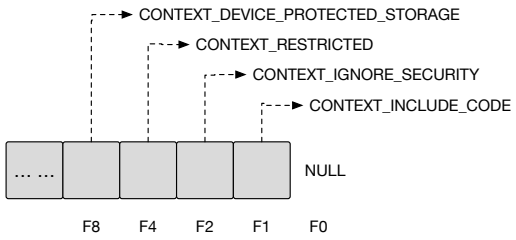


Figure 4: Example options that can be applied when creating contexts via package names.

API parameter checking: There is another constraint that may keep DICI from happening in practice. This constraint is brought by the second parameter of the *createPackageContext()* API. This second parameter, known as *flags*, allows developers to specify (via bitwise operators) how should the package context be created. Some of the options that developers can specify are highlighted in Figure 4 and briefly explained below.

- *F0 (or 0000).* The default option. None of the other options are enabled.
- *F1 (or 0001).* If enabled, **it allows the context to access the code implemented in the loaded package**. Otherwise, only resource files are allowed to be accessed.
- *F2 (or 0010).* This option will ask the context to ignore any security restrictions. When enabled along with *CONTEXT_INCLUDE_CODE*, it will allow code to be loaded into a process even when it is not safe to do so. As recommended by Google, developers should use this option with extreme care[4].
- *F4 (or 0100)*: This option will allow the context to disable specific features of its accessed resources.
- *F8 (or 1000).* This option allows the context to access APIs even at device-protected storage.

[4]https://developer.android.com/reference/android/content/Context

Table 1: DICI-relevant APIs

Level	Class	Signature	Return	Description
1	android.content.Context	createPackageContext(java.lang.String,int)	android.content.Context	used to create a new context object of a specified application. The arguments are application name and creation flags. With flag *CONTEXT_INCLUDE_CODE* and *CONTEXT_IGNORE_SECURITY*, the code of another application can be loaded.
2	android.content.Context	getClassLoader()	java.lang.ClassLoader	get the class loader.
3	java.lang.ClassLoader	loadClass(java.lang.String)	java.lang.Class	get a specified class object by passing the name of the class.
4	java.lang.Class	getConstructor(java.lang.Class[])	java.lang.reflect.Constructor	get the constructor of a class with the argument specifying the signature.
		getDeclaredConstructor(java.lang.Class[])		
4	java.lang.Class	getDeclaredMethod(java.lang.String,java.lang.Class[])	java.lang.reflect.Method	get the method of a class with the arguments specifying the signature.
		getMethod(java.lang.String,java.lang.Class[])		
4	java.lang.Class	getDeclaredField(java.lang.String)	java.lang.reflect.Field	get the field of a class by passing the name.
		getField(java.lang.String)		
5	java.lang.Class	newInstance()	java.lang.Object	instantiate a class with its zero-argument constructor.
5	java.lang.reflect.Constructor	newInstance(java.lang.Object[])	java.lang.Object	instantiate a class with the specified constructor.
5	java.lang.reflect.Method	invoke(java.lang.Object,java.lang.Object[])	java.lang.Object	invoke the method. The first argument specifies the instance of the class and passing *null* indicates a class method.
5	java.lang.reflect.Field	set(java.lang.Object, java.lang.Object)	void	set the field with a certain value. The first argument indicates the object to which the field belongs and *null* means the field is static. For set*(java.lang.Object, *), the asterisk can be replaced with boolean, byte, char, double, float, int, long and short. For example, setInt(int).
		set*(java.lang.Object, *)		
		get(java.lang.Object)	java.lang.Object	get the value of a field. The asterisk stands for the same primary types mentioned in set*(java.lang.Object, *).
		get*(java.lang.Object)	*	
5	java.lang.reflect.Method java.lang.reflect.Field java.lang.reflect.Constructor	setAccessible(boolean)	void	set the accessibility of the method, field or constructor.

In order to invoke the code of other apps, when creating the context via *createPackageContext()*, the F1 option has to be enabled. Therefore, in this step, we further take efforts to trace the value of the *flags* parameter (through backward constant propagation) of located *createPackageContext()* usages. If the F1 option is not enabled, the corresponding API call will not be considered, so as to avoid false-positive results.

3.3 Step (3): Context-aware Flow-sensitive Data-flow Analysis

While a call-graph is relevent to spot call sequences of DICI-relevant APIs, this sequence may actually not be about implementing a DICI. Indeed, APIs may be invoked under different contexts, leading to a situation where there is no actual inter-app interaction. Thus, we need to ensure that the code block that is eventually invoked is indeed reused from another app. Let us consider the example provided in Listing 4. This Listing is similar to the beginning of Listing 1 for the case of malicious code hiding: there is a difference in that the `loader` is instantiated by calling `this.getClassLoader()` rather than `invokee.getClassLoader()`. As a result, although the APIs of the first two levels are invoked following the ideal sequence (i.e., Level-1 method is called before the Level-2 method), they cannot jointly form a DICI as the *loader* is not obtained from the context *invokee* but the current context (i.e., *this*).

```
1 Context invokee = this.createPackageContext(
2   "org.communicorpbulgaria.bgradio",
3   Context.CONTEXT_INCLUDE_CODE |
4   Context.CONTEXT_IGNORE_SECURITY);
5 ClassLoader loader = this.getClassLoader();
6 //ClassLoader loader = invokee.getClassLoader();
```

Listing 4: An example code showing the necessity of taking context into consideration.

We address the challenging of keeping track of the data-flow between API calls by performing a *context-aware data-flow analysis* to ensure that the APIs are all called under the same context. Nevertheless, instead of performing a generic context-aware data-flow analysis, which tracks all the flow of all the variables and hence could be compute-intensive, DICIDer implements a dedicated context-aware data-flow analysis for which only the contexts related to the DICI-relevant APIs are tracked.

3.4 Step (4): DICI Usage Identification

Finally, in the last step, DICIDer leverages the results of the previous steps to pinpoint DICI paths. We recall that the output of step 3 is a DICI path which is a sequence of API calls with a least one API for each defined level and called under the same context. However, at this stage, it is still not established which app, class and method are invoked via DICI, i.e., what is the target code for reuse. We introduce a lightweight *constant string propagation module* in DICIDer, which goes one step deeper to infer what are the methods/fields that are accessed via DICI. To that end, given a fifth-level API, such as *java.lang.reflect.Method.invoke()*, we perform a backward string analysis to infer which is the reflectively-accessed artifact. Regarding the example shown in Listing 1, for the *invoke* method illustrated in Line 16, our backward string analysis aims at infering that the method, which is called via reflection, is *getDeviceID()* of the class *org.ccb.radioapp.components.Utils* in app *org.communicorpbulgaria.bgradio*.

4 EVALUATION

We empirically assess DICIDer, and investigate the use of DICIs in the real-world.

Research questions: The study is driven by the following research questions (RQs).

- **RQ1:** *Can* DICIDer *spot DICIs in real-world Android apps?* To answer this RQ, we investigate on the one hand the recurrence of DICIs in apps collected from various markets. On the other hand, we study the prevalence of DICIs among goodware and malware apps respectively.
- **RQ2:** *How DICI usages evolve over time?* To answer this RQ, we consider both the evolution of number of apps leveraging DICIs within markets, as well as the evolution of DICIs usages within app lineages (i.e., based on their updates).
- **RQ3:** *For what purposes do developers implement DICIs?* We consider a number of real-world examples to dissect the purposes of DICI usages.

Dataset: The evaluation is conducted on apps collected from AndroZoo [17], a continuously growing repository of Android apps. At the time of writing, the dataset size was over 10 million apks crawled from the Google Play official store as well as from alternative markets and repositories. Some metadata on the apps are also collected via the toolkits provided by Li et al. [33].

4.1 RQ1: DICIs in Real-world apps

The goal is to run DICIDer in order to attempt the detection of DICIs in real-world Android apps. To that end, we sample Android apps following their market provenance.

Comparison among Markets. Currently, the top-4 sources ranked based on the number of apps crawled in AndroZoo are Google Play, PlayDrone, Anzhi and AppChina. However, since PlayDrone is a specific subset of apps originally crawled from Google Play, we do not consider PlayDrone as a distinct provenance. Thus, we consider mainly the remaining 3 sources and randomly select 25,000 apps from each provenance[5] leading to a total of 75,000 Android apps.

Table 2: DICI Comparison among Markets

	Google	Anzhi	AppChina	Total
# of successfully analyzed apps	25,000	25,000	25,000	75 000
# of apps with DICIs	4,344	100	135	4579
Percentage of apps with DICIs	17.38%	0.40%	0.54%	6.11%
# of detected DICIs	4,396	1,051	227	5674
Median # of DICIs per App[6]	1	13	1	1

Table 2 provides statistics of the execution of DICIDer on the 75k real-world apps. Overall, DICIDer is able to detect a significant number of DICIs. At the market level, we notice that apps from the official market, *Google Play*, are much more likely to contain DICIs than apps from the alternative markets. A priori, this is reassuring since alternate markets are known to include more malicious samples than the official markets [34]. Nevertheless, when looking at the median number of DICIs per app, apps from market *Anzhi* exhibit a remarkably higher number of DICIs than for other markets when considering apps that implement this reuse mechanism. Figure 5 gives a more concrete understanding of the difference between *Anzhi* and the other markets from the perspective of DICI per app. Further statistical investigations of *Google Play* cases reveal that about 97% of DICIs are from a class named *com.google.android.gms.dynamite.DynamiteModule*.

Google Mobile Services and DICIs. We focus on the *DynamiteModule* class that is recurrently involved with DICIs of GooglePlay apps. Based on its package name, we suspect that it is may be part of the official *Google Mobile Service* (GMS) APIs. The official documentation does not however mention such a package. We postulate that such a package may have been intentionally omitted from the documentation to avoid uses by third-party developers. Nevertheless, we undertake to confirm the presence of this class within GMS by explicitly requesting Gradle dependency management to find the GMS libraries and included them in a toy/demo app. Afterwards, we manually analyzed the content of the class to further check what it does through DICI. According to the analysis report of DICIDer, *DynamiteModule* code instantiates a class named *com.google.android.gms.dynamite.IDynamiteLoader* from the app named *com.google.android.gms*. To further check how this instance is used, we proceed to reverse-engineer an app that contains GMS APIs the code of such APIs are not open-sourced. According to the decompiled code, this class implements the interface *android.os.IBinder* which is designed for in- and cross- process calls[7] and is used to query a local interface here. Although this class is also under the package of GMS *dynamite* according to its name. It cannot be found in the GMS libraries. Since there is quite few information about these libraries. We can only infer that the app *com.google.android.gms* is the GMS framework which is supposed to be embedded into the Android OS, and direct inter-app code invocation is the way to access the framework.

We also consider class *org.xwalk.core. XWalkCoreWrapper* which contributes to most DICIs in Anzhi dataset. Class *XWalkCoreWrapper* is from a project called *CrossWalk* which was once founded by *Intel's Open Source Technology Center*[8]. It is a web app runtime to provide manipulability to browser. The class uses DICIs to access functionalities of its own app.

While we studied the recurrent cases in this RQs, we will consider the remaining 3% in Google Play datasets for answering RQ3.

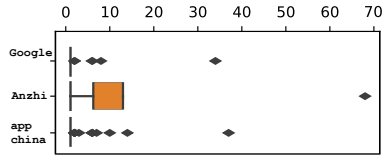


Figure 5: Distribution of DICI per App

Comparison between Benign Apps and Malware. AndroZoo not only crawls Android apps, but also the antivirus reports from VirusTotal [35] for each app. For a given app, AndroZoo indicates the number of anti-virus products which flag the app as a malware (among a total of about 60 anti-virus products). We rely on this information to build our dataset of goodware and malware. For goodware, we consider 25,000 apps selected from AndroZoo with no flag from any anti-virus product. For malware, we consider 25,000 apps selected from AndroZoo with at least 30 flags (i.e., at least half of the anti-virus have a consensus on app maliciousness).

[5] Since DICIDer may fail to analyze some apps due to unexpected corner cases such as the given APK does not contain DEX file, in practice, we have randomly listed all apps and sequentially tested them until the quota of 25000 is reached for each provenance.

[7] According to official document at https://developer.android.com/reference/android/os/IBinder

[8] See project page at https://crosswalk-project.org/

Table 3 presents the comparison between Goodware and Malware. Surprisingly, overall, malware actually use much less DICIs compared to benign apps. However, by further checking the source package of DICIs, we find that for benign apps, the dominated class is again *com.google.android.gms.dynamite.DynamiteModule* While, for malware, this class only contributes a quarter of DICI usages. This has two implications: the scope of using DICI for benign scenarios is still limited, although many instances of benign apps, because of their reliance on GMS, are actually hosting code that use the DICI mechanism; malicious apps on the other hand may have indeed been leveraging DICIs.

Table 3: DICI Comparison between Goodware and Malware

	Benign	Malware
# of successfully analyzed apps	25,000	25,000
# of apps with DICIs	5,836	52
Percentage of apps with DICIs	23.34%	0.2%
# of detected DICIs	5,964	101
Median # of DICIs per App[9]	1	1

Answer to RQ1: DICIDer is able to detect DICIs in real-word apps. This reuse mechanism is actually seen in many apps, although mostly due to the use of the GMS libraries where class *com.google.android.gms.dynamite.DynamiteModule* heavily relies on DICI. There are however cases of malware leveraging DICI outside the scope of GMS libraries.

4.2 RQ2: Evolution of DICI Usages.

As shown in RQ1, *com.google.android.gms.dynamite.DynamiteModule* is the major source of DICIs. We noticed that this class was not present in the Android ecosystem since the beginning of Android. Thus, we propose to study the evolution in time of the number of DICIs within Android apps. To perform this experiment, we consider app lineages (i.e., different versions of apps over time). To that end, we consider a large lineage dataset proposed by Gao et al. [36] based on the AndroZoo repository. Most of the lineages are spread over several years, but for each year, we consider only the latest apk version in that year for a given lineage. The statistics of apks per year from the lineage dataset in the literature is listed in Table 4.

Table 4: # APKs considered from the Lineage dataset [36]

2011	2012	2013	2014	2015	2016	2017	2018
3,950	6,252	13,191	23924	17,505	30,690	6,995	3,583

We run DICIDer on this dataset and compute the percentage of apks containing DICIs for each year. The result is presented in Figure 6. A clear increasing trend can be observed after year 2015. By further investigating the DICI contributors, we notice that the main reason is still the GMS libraries. We find that before 2016, there are no contributors from GMS libraries. However, starting from 2016, the main contributors are all from GMS libraries, although the source packages shift from *internal* (mainly in 2016) to *dynamite* (after 2016). Unfortunately, these library not being part of the Android Open Source Project, we cannot find any information about the APIs publication and update time. We infer however that, starting from 2016, GMS libraries start to be more and more used Android applications.

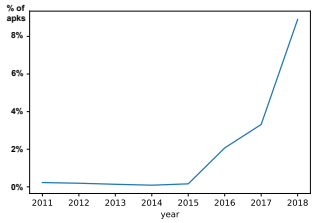


Figure 6: Percentage of APKs Contain DICI

By leveraging lineage, we also have the chance to investigate "update" patterns through checking the status of DICI usages in different versions of a same app. For this experiment, we consider all lineages with at least one DICI present in at least one apk of the lineage, and lineage with at least 4 apks. Table 5 presents the distribution of different update patterns. In a large majority of the cases, DICIs are not implemented in the initial version of the app. Instead it is added during an update.

Table 5: # APKs of Each Year

% of lineages in which a DICI has been introduced by an update	89.38%
% of lineages in which a DICI has been removed after an update	8.30%
% of lineages in which a DICI has been in all versions	2.07%
% of lineages with DICI added, then removed, and then added	0.26%

Answer to RQ2: Over time, the use of the DICI mechanism has progressively become boomed among Android apps, mainly due to the availability of the GMS library. Nevertheless, it is noteworthy that DICIs can be implemented during app updates, which may cause concerns for update attacks, since it is well-known that users are less wary of apps during updates [37, 38].

4.3 RQ3: Purposes of Using DICIs

Tables 6 and 7 enumerate respectively the top 10 DICI contributor packages where DICIs are invoked and the top 9 apps that are targeted by DICI reuse cases. These statistics are based on the lineage dataset described above. We note that, besides the fact that a GMS package is a top contributor for DICI usage, the GMS app is also the top app whose code is largely targed for direct inter-app invocation. From the package and app names, we can notice the connection between some of them such as package *com.jb.gosms.util* tries to access code in app *com.jb.gosms.emoji*. Actually, by manually checking all the top 10 contributors, we confirm that they all try to access apps with similar names. For these cases, we can infer that they try to access the code from the app owned by same developers, to perform *app collusion* scenarios.

Table 6: Top DICI Contributor Packages

Rank	Package	Number
1	com.google.android.gms	7,462
2	com.lbe.doubleagent.client	128
3	com.jb.gosms.util	37
4	net.pierrox.lightning_launcher.b	30
5	com.dokdoapps.utility.GoogleServiceManager	20
5	com.handcent.common.v	20
7	kl.ime.oh.H	14
8	com.google.android.apps	13
9	org.xwalk.core.ReflectionHelper	12
10	cn.longmaster.common.pluginfx	10

Table 7: Top DICI Invoked Apps

Rank	Package	Number
1	com.google.android.gms	7,469
2	com.jb.gosms.emoji	37
3	klye.hanwriting	14
4	org.xwalk.core	12
5	net.pierrox.lightning_locker_p	2
5	com.pansi.msg.plugin.custom_notify	2
6	com.pansi.msg.plugin.regins	1
6	com.shocktech.guaguahappy	1
6	com.pansi.msg.plugin.emoji	1

We take one more step to reveal the purpose of apps using DICIs by deeply exploring some apps.

Plugin implementation through DICI. The app with package name *kl.ime.oh*[10] is a multi-language keyboard app developed by *Honso* with more than 1 million installs on *Google Play*. It is found using 5 methods from another app with package name *klye.hanwriting*, which is a Chinese keyboard plugin app also can be found on *Google Play* with the same developer[11]. By searching apps from the same developer, more plugins for other languages can be found as well. We further notice that when the code loading failed, the app will return a string to ask to "Download Chinese plugin". Thus, we can infer that this app implemented a plugin functionality by using the DICI code reuse mechanism. We found several apps performing similar plugin behaviour. App named *com.jb.gosms* is found with loading 13 methods from an app with package name *com.jb.gosms.emoji* and a *BroadcastReceiver* is registered to check if the app is newly installed or uninstalled. App *com.pansi.msg* loads code from 3 different apps which are *com.pansi.msg.plugin.custom_notify*, *com.pansi.msg.plugin.emoji* and *com.pansi.msg.plugin.regins*. And app *net.pierrox.lightning_launcher* loaded code from *net.pierrox.lightning_locker_p*. We further notice that for the value of flags when creating the app context, some of them used value 1 instead of 3 which omits the flag of *CONTEXT_IGNORE_SECURITY*. This could be because all these apps are from the same developer (i.e., containing same signatures). Nevertheless, it is an interesting feature to be considered by static analyzers when assessing the security risks.

Keeping up with best practices through GMS. We find another use of the DICI code reuse mechanism which is to load up-to-date functionalities matching the best practices of Android programming available via *com.google.android.gms*. *GMS* stands for *Google Mobile Services* and *com.google.android.gms* is the *Google Play Services Packages*. The loader apps try to invoke the method *insertProvider* from class *com.google.android.gms.common.security.ProviderInstallerImpl*. According to the official documentation from *Google*[12], the purpose is to update the security provider to protect against SSL exploits. However, the relevant method mentioned in the document is *installIfNeeded* which is different from the one we found. By further checking the source code of method *installIfNeeded*, we find that the fundamental method is also *insertProvider*. Some developers may also notice this. Thus, instead of invoking the documented method and include all relevant libraries, they chose to directly invoke the fundamental method. All these loader apps are with package name of *com.monese.monese.live*, *nya.miku.wishmaster* and *com.levelup.touiteur* respectively[13].

5 COUNTERMEASURES

We now discuss possible countermeasures that could be leveraged by app developers to protect their app code from being reused in a stealthy way through DICI.

We found a straightforward countermeasure that, until now, we could not find a way to break. The idea is to check, with the code to protect, what is the "instance" of the application that is executing it. Indeed, to the best of our knowledge, there is no means to get or generate an instance of another app. Listing 5 presents the code that a developer could use to protect her app. First, to record the app's instance, a slight modification to the Application class is required (here `AntiTheftApp`). Specifically, we create a private instance field (line 2) and assign the current instance to this field (line 7). Then, we create an empty method called *verify* (lines 14 to 15). The purpose of this method is to check the availability of the stored application instance (i.e., *theInstance* at line 2), and an empty method is already enough. Indeed, when `getInstance().verify()` is called in the original app, nothing happens. However, when this method is called in the plagiarist app, the app will crash because `theInstance` has not been initialized, and yet it cannot be instantiated or overwritten by attacking code in other apps Finally, to protect from being reused (via DICI), app developer can simply write `AntiTheftApp.getInstance().verify()` at the beginning of each method, she wants to protect.

```
1 public class AntiTheftApp extends Application {
2     private static AntiTheftApp theInstance;
3     @Override
4     public void onCreate() {
5         super.onCreate(); theInstance = this;
6     }
7     public static AntiTheftApp getInstance() {
8         return theInstance;    }
9     public void verify() {  }
```

Listing 5: Implementation of Application Instance based Verification

Other Possibilities: So far, we have only attempted to protect DICIs at the app code level. Yet we believe many other features could also be leveraged to protect DICIs. Indeed, on the one hand, native libraries (or Javascript code for WebKit-based apps) could be leveraged, as they increase significantly the complexity of the code, making it non-trivial to be bypassed by attackers. On the other hand, some system features could be leveraged as well. For example, each Android app will be allocated with a private directory that cannot be accessed by other apps, and thereby could be leveraged to check the identity of the active app. Last but not least, from the Android framework point of view, there are various countermeasures could be applied. For example, Android OS could provide a mechanism similar to permission and component management for the inter-app code invocation functionality. It could limit the code access within apps from the same developers, or allow a declaration in the AndroidManifest file to specify which part of the code (i.e., classes) can be accessed publicly by other apps through DICIs. It also can be limited based on the privileges of Linux users.

[10]It can be downloaded from *AndroZoo* by using the SHA256: 04E37D1CE54C7E326A7714F56B35F922DF9EAF5AAD190FC5FD61716F84176D3E and the app can be found on *Google Play* with link: https://play.google.com/store/apps/details?id=kl.ime.oh

[11]The page link is: https://play.google.com/store/apps/details?id=klye.hanwriting

[12]https://developer.android.com/training/articles/security-gms-provider

[13]The SHA256s are: E953776572E4E84CB64D0ABF442211FC9A5EDDF0BDEF7E5DFC47C94756C714AB, 8B16DBD2D4951BDAB16F2AA9AABCBC8BEE91264DAB78F2BCAFD3EE317B84E27C, 06E280B615D5CF68E6BD3F89E27FF11FDBCFCE7D038AACCFE87C486F47159EB6

6 LIMITATIONS

The fact that our prototype tool has revealed various DICI usages in real-world Android apps shows that our tool is useful to pinpoint them. Nonetheless, the implementation of our tool has come with various limitations. First of all, since the dummy main method construction approach is borrowed from FlowDroid, all the relevant limitations reported by FlowDroid also apply to our approach. For example, unsoundness can arise if certain callbacks in Android lifecycles are overlooked when building the dummy main method. Second, DICIDer directly adopts the constant propagation approach provided by Soot which unfortunately only supports intra-procedural analysis. As a result, although it is not our main focus of this work, certain reflectively accessed methods or fields could be missed by our approach. We keep this for future work. At the moment, DICIDer does not take into account native libraries and is oblivious to multiple-threading implementations, which may result in unsound results as well.

Not only the implementation of our prototype tool comes with limitations, the validity of our experimental results may also be threatened by the experimental setup we designed in this work. The major threat to the validity lies in the choice of selected Android apps. Although we rely on a random selection from AndroZoo to prepare the real-world apps for analysis, since the distributions of apps in different markets available in AndroZoo vary significantly, we cannot guarantee the representativeness of these apps. Furthermore, we leverage the app assembly time to build app lineages in this work. The app assembly time, as experimentally revealed by Li et al. [39], may not be accurate to represent the app release time. Hence, the app lineages we leverage to study the evolution of DICI usages may not be reliable as well. In this work, we try to mitigate this by following the same approach of our fellow researchers to build the app lineages, which have been demonstrated to be useful to support app evolution studies.

7 RELATED WORK

To the best of our knowledge, this paper presents the first work disclosing the possibility of direct inter-app code invocation among Android apps and subsequently detecting DICI usages in Android apps. As a result, there is no related work specifically focusing on this problem. However, the research community has proposed various contributions in the domain of static analysis of Android apps. Moreover, some works focused on the problems of Inter-Component Communication (ICC) and Inter-App Communication (IAC), which are closely related to DICI. We now discuss the representative ones.

Static Analysis of Android Apps: Many state-of-the-art works have adopted static analysis, as one of their fundamental parts, to perform their research investigations. As presented in a recent survey done by Li et al. [40], there are over 100 papers, published mainly in the software engineering and security community, proposed to analyze Android apps statically. As revealed in their survey, static analysis has been largely conducted to uncover security and privacy issues such as privacy leaks detection [10, 27] and malware detection [41, 42]. Also, the survey discloses that the well-known Soot framework is the most adopted basic support tool in the community to implement static analysis approaches. We remind that the Soot framework is also leveraged by DICIDer to detect the usage of DICIs. Static analysis has also been used by researchers to scan for (1) app defects including energy issues [43, 44], (2) fix runtime crashes [45, 46], (3) improve the realization of dynamic testing appraoches [47–50].

Focus on Inter-Component Communication: Android apps differ from traditional Java apps in that there is no single entry point, e.g., the main method, in the apps. Apps are composed of multiple basic components. To pass on data among these components, Android has a special Inter-component Communication (ICC) mechanism. However, malware may also use this mechanism to achieve their malicious behaviors, e.g., steal users' private data. To this end, our community has proposed various approaches to mitigate the attacks related to Android ICCs. As an example, Epicc [9] is proposed to reduce the ICC problem to an instance of the Interprocedural Distributive Environment (IDE) problem, and finds ICC vulnerabilities with far fewer false positives. IccTA [10] is a static taint analyzer to detect privacy leaks among components in Android applications. It goes beyond existing ICC leaks detection tools like [11–13].

Focus on Inter-Application Communication: Android's Inter-application communication (IAC) mechanism allows for reuse of functionality across apps via *Intent*s. Contrary to the technique described in this paper, IAC is intended for functionality sharing. However, this mechanism also raises concerns for vulnerabilities crossing Android apps. Thereby, the research community has also proposed various approaches to mitigate possible vulnerabilities brought by IAC. For example, Li et al. [21] have proposed a tool called ApkCombiner aiming to combine multiple apps together to a single app, so as to reduce an IAC problem to an ICC problem. As a result, this tool allows all the aforementioned ICC-aware approaches to resolving IAC problems without modifications. PermissionFlow [51] can reliably and accurately detect vulnerable information flows among Android applications. IntentDroid is a cloud-based testing algorithm for Android apps for automated discovery of Android IAC vulnerabilities. ComDroid [52] is another tool to detect ICC related malicious behaviors in Android apps, e.g., sniffing message contents and injecting forged messages.

Unfortunately, since DICI leverages a totally different channel to implement inter-app communication, all the aforementioned existing works cannot be directly applied to detect DICI usages in Android apps. Our prototype tool DICIDer fills this gap by providing a means to statically pinpoint DICI usages in Android apps, which could be considered as a complement to the state-of-the-art.

8 CONCLUSION

In this paper, we disclose to the software engineering community a novel mechanism allowing direct inter-app code invocation (DICI) among installed Android apps on mobile devices. Through concrete motivating examples, we demonstrate that DICI can be leveraged to successfully perform malicious attacks and plagiarize the core function of the competitor's apps. We then introduce to the community a static analyzer called DICIDer to automatically locate the usage of DICIs in Android apps. Experiments on a large set of Android apps reveal that DICIDer is indeed capable of detecting DICIs in Android apps, and the usage of DICIs tends to increase over time, which may cause concerns for update attacks since users might be less wary of apps during updates.

REFERENCES

[1] William B Frakes and Kyo Kang. Software reuse research: Status and future. *IEEE transactions on Software Engineering*, 31(7):529–536, 2005.

[2] Israel J Mojica Ruiz, Meiyappan Nagappan, Bram Adams, and Ahmed E Hassan. Understanding reuse in the android market. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 113–122. IEEE, 2012.

[3] Li Li, Tegawendé F Bissyandé, and Jacques Klein. Rebooting research on detecting repackaged android apps: Literature review and benchmark. *IEEE Transactions on Software Engineering*, 2019.

[4] Jin-Hyuk Jung, Ju Young Kim, Hyeong-Chan Lee, and Jeong Hyun Yi. Repackaging attack on android banking applications and its countermeasures. *Wireless Personal Communications*, 73(4):1421–1437, 2013.

[5] Wu Zhou, Xinwen Zhang, and Xuxian Jiang. Appink: watermarking android apps for repackaging deterrence. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 1–12, 2013.

[6] Mu Zhang and Heng Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *NDSS*, 2014.

[7] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei 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, 2012.

[8] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 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)*, pages 543–558, Washington, D.C., 2013. USENIX.

[9] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 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)*, pages 543–558, 2013.

[10] L. Li, A. Bartel, T. F. Bissyandé, 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*, volume 1, pages 280–291, May 2015.

[11] Jonathan Burket, Lori Flynn, William Klieber, Jonathan Lim, Wei Shen, and William Snavely. Making didfail succeed: Enhancing the cert static taint analyzer for android app sets. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA PITTSBURGH United States, 2015.

[12] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. Scandroid: Automated security certification of android. Technical report, 2009.

[13] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. The impact of vendor customizations on android security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 623–634, 2013.

[14] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, volume 30, page 88, 2011.

[15] Tristan Ravitch, E Rogan Creswick, Aaron Tomb, Adam Foltzer, Trevor Elliott, and Ledah Casburn. Multi-app security analysis with fuse: Statically detecting android app collusion. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, pages 1–10, 2014.

[16] Amiangshu Bosu, Fang Liu, Danfeng Yao, and Gang Wang. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 71–85, 2017.

[17] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM.

[18] Karim O Elish, Haipeng Cai, Daniel Barton, Danfeng Yao, and Barbara G Ryder. Identifying mobile inter-app communication risks. *IEEE Transactions on Mobile Computing*, 19(1):90–102, 2018.

[19] Shweta Bhandari, Wafa Ben Jaballah, Vineeta Jain, Vijay Laxmi, Akka Zemmari, Manoj Singh Gaur, Mohamed Mosbah, and Mauro Conti. Android inter-app communication threats and detection techniques. *Computers & Security*, 70:392–421, 2017.

[20] Damien Octeau, Somesh Jha, Matthew Dering, Patrick Mcdaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *Proceedings of the 43th Symposium on Principles of Programming Languages (POPL 2016)*, 2016.

[21] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Apkcombiner: Combining multiple android apps to support inter-app analysis. In *IFIP International Information Security and Privacy Conference*, pages 513–527. Springer, 2015.

[22] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics and Security*, 12(6):1269–1284, 2017.

[23] TikTok. Web site: https://www.tiktok.com.

[24] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, Junbum Shin, and SWRD Center. Scandal: Static analyzer for detecting privacy leaks in android applications. *MoST*, 12(110):1, 2012.

[25] Christopher Mann and Artem Starostin. A framework for static detection of privacy leaks in android applications. In *Proceedings of the 27th annual ACM symposium on applied computing*, pages 1457–1462, 2012.

[26] Siddharth Prakash Rao, Silke Holtmanns, Ian Oliver, and Tuomas Aura. Unblocking stolen mobile devices using ss7-map vulnerabilities: Exploiting the relationship between imei and imsi for eir access. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 1171–1176. IEEE, 2015.

[27] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 49(6):259–269, June 2014.

[28] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, volume 14, page 1125. Citeseer, 2014.

[29] Li Li, Tegawendé F. Bissyandé, Damien Octeau, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 318–329, New York, NY, USA, 2016. Association for Computing Machinery.

[30] P. Calciati and A. Gorla. How do apps evolve in their permission requests? a preliminary study. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 37–41, May 2017.

[31] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. Permission evolution in the android ecosystem. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 31–40, New York, NY, USA, 2012. Association for Computing Machinery.

[32] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.

[33] Li Li, Jun Gao, Médéric Hurier, Pingfan Kong, Tegawendé F. Bissyandé, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Androzoo++: Collecting millions of android apps and their metadata for the research community, 2017.

[34] Haoyu Wang, Zhe Liu, Jingyue Liang, Narseo Vallina-Rodriguez, Yao Guo, Li Li, Juan Tapiador, Jingcun Cao, and Guoai Xu. Beyond google play: A large-scale comparative study of chinese android app markets. In *The 2018 Internet Measurement Conference (IMC 2018)*, 2018.

[35] VirusTotal. Web site: https://www.virustotal.com/.

[36] J. Gao, L. Li, P. Kong, T. F. Bissyandé, and J. Klein. Understanding the evolution of android app vulnerabilities. *IEEE Transactions on Reliability*, pages 1–19, 2019.

[37] Francesco Mercaldo, Vittoria Nardone, Antonella Santone, and Corrado Aaron Visaggio. Download malware? no, thanks: how formal methods can block update attacks. In *Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering*, pages 22–28, 2016.

[38] David Barrera, Jeremy Clark, Daniel McCarney, and Paul C Van Oorschot. Understanding and improving app installation security mechanisms through empirical analysis of android. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 81–92, 2012.

[39] Li Li, Tegawendé F Bissyandé, and Jacques Klein. Moonlightbox: Mining android api histories for uncovering release-time inconsistencies. In *The 29th IEEE International Symposium on Software Reliability Engineering (ISSRE 2018)*, 2018.

[40] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 2017.

[41] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian 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, 2012.

[42] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *2012 Seventh Asia Joint Conference on Information Security*, pages 62–69. IEEE, 2012.

[43] Haowei Wu, Shengqian Yang, and Atanas Rountev. Static detection of energy defect patterns in android applications. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 185–195, 2016.

[44] Luis Cruz, Rui Abreu, John Grundy, Li Li, and Xin Xia. Do energy-oriented changes hinder maintainability? In *The 35th IEEE International Conference on Software Maintenance and Evolution (ICSME 2019)*, 2019.

[45] Pingfan Kong, Li Li, Jun Gao, Tegawendé F Bissyandé, and Jacques Klein. Mining android crash fixes in the absence of issue-and change-tracking systems. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 78–89, 2019.

[46] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. Repairing crashes in android apps. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 187–198. IEEE, 2018.

[47] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*, 68(1):45–66, 2018.

[48] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 94–105, 2016.

[49] Hailong Zhang, Haowei Wu, and Atanas Rountev. Automated test generation for detection of leaks in android applications. In *Proceedings of the 11th International Workshop on Automation of Software Test*, pages 64–70, 2016.

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

[51] Dragos Sbîrlea, Michael G Burke, Salvatore Guarnieri, Marco Pistoia, and Vivek Sarkar. Automatic detection of inter-application permission leaks in android applications. *IBM Journal of Research and Development*, 57(6):10–1, 2013.

[52] Roee Hay, Omer Tripp, and Marco Pistoia. Dynamic detection of inter-application communication vulnerabilities in android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 118–128, 2015.