ApkCombiner: Combining Multiple Android Apps to Support Inter-App Analysis

Li Li[1], Alexandre Bartel[2], Tegawendé F. Bissyandé[1], Jacques Klein[1], Yves Le Traon[1]

[1] Interdisciplinary Center for Security, Reliability and Trust (SnT), University of Luxembourg
[2] EC SPRIDE, Technische Universität Darmstadt
firstName.lastName@uni.lu, firstName.lastName@ec-spride.de

Abstract. Android apps are made of components which can leak information between one another using the ICC mechanism. With the growing momentum of Android, a number of research contributions have led to tools for the intra-app analysis of Android apps. Unfortunately, these state-of-the-art approaches, and the associated tools, have long left out the security flaws that arise across the boundaries of single apps, in the interaction between several apps. In this paper, we present a tool called `ApkCombiner` which aims at reducing an inter-app communication problem to an intra-app inter-component communication problem. In practice, `ApkCombiner` combines different apps into a single apk on which existing tools can *indirectly* perform inter-app analysis. We have evaluated `ApkCombiner` on a dataset of 3,000 real-world Android apps, to demonstrate its capability to support static context-aware inter-app analysis scenarios.

1 Introduction

Everyday, millions of users exploit their handheld devices, such as smartphones, for online shopping, social networking, banking, email, etc. At the Google I/O 2014, it was revealed that there are now more than 1 billion active Android users and over 50 billion app downloads so far. Thus, mobile applications are increasingly playing an essential role in our daily life, making the safety guards in mobile operating systems an important concern for researchers and practitioners. Because the Android OS accounts for more than 80% of the global smartphone shipments, it has become a primary target of hackers who are now developing malicious apps at an industrial scale [1]. Kaspersky has even reported in a recent security bulletin that, now, 98% of mobile malware found target the Android platform.

An Android app is a combination of components that use a special interaction mechanism to perform Inter-Component Communication (ICC). This light communication model has been exploited by developers to design rich application scenarios by reusing existing functionality. Unfortunately, because many Android developers have limited expertise in security, the ICC mechanism has

brought a number of vulnerabilities [4, 18]. Examples of known ICC vulnerabilities [3] include the *Activity Hijacking* vulnerability (where a malicious *Activity* is launched in place of the intended *Activity*) and the *Intent spoofing* vulnerability (where a malicious app sends Intents to an exported component which originally does not expect Intents from that app). In previous work [17], we have shown that Android components can exploit such ICC vulnerabilities to leak private data. More recent works have further demonstrated that Android apps exhibit various privacy leaks that are built around the ICC mechanism [13, 16, 20].

The privacy leaks in Android are further exacerbated by the fact that several applications can interact and "collaborate" to leak data using the inter-app communication (IAC) mechanism. IAC and ICC are similar in Android, and thus present the same vulnerabilities. Unfortunately, state-of-the-art analysis tools are focused on ICC by analyzing a single app at a time. Consequently, inter-app privacy leaks cannot be identified and managed by existing tools and approaches from the literature.

In this paper we propose to empower existing static analysis tools for Android to work beyond the boundaries of a single app, so as to highlight security flaws in the interactions between two or more apps. To that end we have designed and developed a tool called `ApkCombiner` which takes as input several apps that may cohabit in the same device, and yields a single app package (i.e., apk) combining the different components from the different apps. The resulting package is ensured to be ready for analysis by existing tools. Thus, since the IAC mechanism is the same as the ICC mechanism, by combining apps, `ApkCombiner` reduces an IAC problem to an ICC problem, allowing existing tools to indirectly perform inter-app analysis without any modification.

During the combination of multiple Android apps, some classes may conflict with one another. In this paper, we take into account two types of conflict: 1) the conflicted classes are exactly same (same name and same content), we solve this type of conflicts by simply dropping the duplicated classes and 2) the conflicted classes are different (same name but different content), we solve this type of conflicts by first renaming the conflicted classes, and then ensuring that all dependencies and calls related to those classes are respected throughout the app code.

The contribution of this paper are as follows:

- We discuss the need for tools to support inter-app analysis, and present a non-intrusive approach that can be leveraged by existing tools which are focused on intra-app analysis.
- We provide a prototype implementation of `ApkCombiner`[4], using an effective algorithm to solve different conflicts which may arise during the combination of multiple Android apps into one.
- We propose an evaluation of `ApkCombiner` on both a motivating example and on a dataset of real-world Android apps. The experimental results show

[3] Refer to Section 2.1 for the concept of component, Activity and Intent in Android.

[4] We make available our full implementation, along with the experimental results at: `https://github.com/lilicoding/ApkCombiner`

that state-of-the-art intra-app analyzers can efficiently leverage our approach to indirectly perform inter-app analyses.

2 Background and Motivation

In this section we first briefly introduce different concepts that are specific to Android (cf. Section 2.1). Then, we motivate our work by highlighting the limitations of state-of-the-art static analysis approaches targeting the Android system (cf. Section 2.2). Finally, we discuss in Section 2.3 an IAC vulnerability through a running example.

2.1 Android IAC Overview

In Android, the inter-component communication (ICC) mechanism allows two components to exchange data and invoke each other. The Android inter-app communication (IAC) mechanism works in the same way and exploits the ICC mechanism to make components from different apps interact. An ICC is typically triggered by one of several specific Android methods which are related to the different components in presence (i.e., either an `Activity, Service, Content Provider, Broadcast Receiver`). Those methods[5] take as parameter a special kind of object, called Intent, which specifies the target component(s), either explicitly, by setting the name of the target component's class, or implicitly, by setting the action to perform, the category and the input data. Obviously, since it is hard for developers to predict which other apps will be available at the same time, IAC invocations are almost always performed through implicit Intents. In order to receive implicit Intents, target components in separate apps need to declare their capabilities, through an Intent Filter, in the app manifest file so that the Android system may match them when requested by a given app.

2.2 Static Analysis for Android Apps

Static program analysis has been widely used to address security issues, e.g., related to data integrity and confidentiality of information flow [7], as well as for anomaly detection [11,19]. More recently, static analysis techniques have also been applied for dissecting Android applications [9,12,14]. However, we note that current approaches still present a number of limitations when they are targeted to code from the Android system.

In the most common case, a static analysis of Android is reduced to an intra-app analysis, where the bytecode of an app is extracted, and parsed to produce a control-flow graph (CFG) to further perform specific analysis. For example, FlowDroid [2], a recent state-of-the-art Android analysis approach, builds CFG for static taint analysis. When the analysis must take into account the interaction between two or more apps, it is refered to as an inter-app analysis and may operate in three different ways as illustrated in Fig. 1.

[5] Except `Content Provider` related methods.

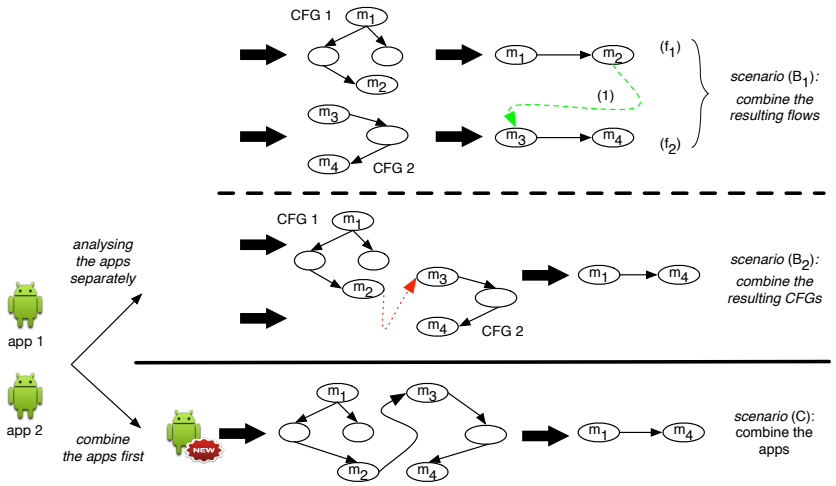


Fig. 1: Different approaches for Enabling Inter-App analysis of Android apps.

At a high livel, there are two options for enabling inter-app analysis: (1) perform intra-app analysis of each app independently from the others and rely on these analysis results to infer inter-app analysis output; or (2) combine the apps first before performing the analysis.

In scenario (B_1), the results of the intra-app analyses (i.e., flows f_1 and f_2) are combined to yield a potential flow between the apps. However, because the combination is performed after the analysis, no context data (e.g., variable values such as data handled by the Intents) is available, and thus the scenario requires to approximate the flows between the apps (e.g., here line (1)). This scenario may thus lead to a significant number of false positives.

Scenario (B_2) is an improved version of (B_1) where it is no longer the results that are combined but the CFGs instead. This scenario thus supports a context-aware inter-app analysis by operating on a combined CFG and on a data dependence graph (DDG) between the apps. Nevertheless, such an approach cannot be generalized to any instances of CFG. In practice for example, the workload for combining CFGs generated by Soot [15] can not even be applied in the case of CFGs generated by Wala[6]. Thus, specific development effort must be put into each and every static intra-app analyzer to support inter-app analysis.

Scenario (C) considers the caveats of all previous approaches by further improving scenario (B_2) to yield a **general** approach for enabling **context-aware** *inter-app* analysis of Android apps. Thus, instead of combining separate CFGs from different apps as in scenario (B_2), the approach would consist in combining the complete apps at the bytecode level. The generated single app package is thus immediately ready for analysis with static intra-app analysis tools. The analysis results will then contain information that could only be obtained through inter-app analysis. This paper presents the design and implementation of a tool for

[6] http://wala.sourceforge.net/wiki/index.php

supporting such an approach where no modification of current state-of-the-art tools will be required to perform inter-app analyses.

Table 1 summarizes the fact that most state-of-the-art static analysis approaches for Android only deal with intra-app analysis. DidFail [13], the only approach that considers inter-app analysis, falls under scenario (B_1) described above. Yet, as reminded by the classification in Table 2, this scenario leads to a context-unaware analysis, and thus to many false positives in the results.

Table 1: State-of-the-art approaches

Intra-App(Inter-Comp)	Inter-App
IccTA [16]	DidFail [13]
AmanDroid [20]	
ScanDroid [10]	
SEFA [21]	
CoChecker [5]	

Table 2: Classification of scenarios for static inter-app analysis approaches. (B_1), (B_2) and (C) refer to the scenarios illustrated in Fig. 1.

	Non-General	General
Context-Unaware	(B_1)	
Context-Aware	(B_2)	(C)

2.3 A Running Example

Fig. 2 presents a running example that shows an IAC vulnerability. The example is extracted from a test case of DroidBench[7] refered among its list as *InterAppCommunication_sendBroadcast1*. The example consists of two Android applications, refered to as *sendBroadcast1_source* and *sendBroadcast1_sink*.

App1: sendbroadcast1_source

```
1: class OutFlowActivity extends Activity{
2: protected void onCreate(Bundle b) {
3: //tm = default TelephonyManager;
4: String imei = tm.getDeviceId();
5: Intent i = new Intent();
6: i.setAction("lu.uni.serval.iac_sendbroadcast1.ACTION");
7: i.putExtra("DroidBench", imei);
8: sendBroadcast(i); }}
```

App2: sendbroadcast1_sink

```
11: class InFlowActivity extends Activity
12: {
13: protected void onCreate(Bundle b) {
14: Intent i = getIntent();
15:  String imei = i.getStringExtra("DroidBench");
16:  Log.i("DroidBench", imei);
17: }}
```

```
21: <activity android:label="@string/app_name" android:name="lu.uni.serval.iac_sendbroadcast1_sink.InFlowReceiver">
22:  <intent-filter>
23:   <action android:name="lu.uni.serval.iac_sendbroadcast1.ACTION" />
24:   <category android:name="android.intent.category.DEFAULT" />
25:  </intent-filter>
26: </activity>
```

App2: AndroidManifest

Fig. 2: A running example that shows an inter-app vulnerability.

App *sendBroadcast1_source* contains a simple *Activity* component, named `OutFlowActivity`, which first obtains the device ID (line 4) and then stores it into an Intent (line 7) which is then forwarded to other components (potentially in other applications since the Intent is implicit (line 6)). App *sendBroadcast1_sink* contains a component called `InFlowActivity`, which first extracts data from the received Intent and then logs it into disk.

[7] DroidBench is a set of hand-crafted Android apps used as a ground truth dataset to evaluate how well static and dynamic security tools find data leaks. `https://github.com/secure-software-engineering/DroidBench`

In this example, we consider device ID, which is protected by a permission check of the Android system, to be sensitive data. We also consider the *log()* method to be dangerous behavior since it writes data into disk, therefore leaving it accessible to any applications, including anyone which does not have permission to access the device ID through the Android OS. Thanks to the declarations in the *Manifest* file in the package of *sendBroadcast1_sink*, `OutFlowActivity` is able to communicate with `InFlowActivity` using the *sendBroadcast()* ICC method. Thus, through the interaction between these two apps, a sensitive data can be leaked.

Unfortunately, the current state-of-the-art static analysis tools, including FlowDroid and IccTA, cannot tackle the kind of IAC problem described above. Since these tools have already proven to be efficient in statically identifying bugs and leaks across components inside a single app, we aim at enabling them to do the same across applications. We further put a constrain on remaining non intrusive, i.e., to avoid applying any modification on them, so as to avoid introducing limitations or new bugs in these tools. We thus propose `ApkCombiner`, which, by combining multiple apps into one, reduces the IAC problem to an ICC problem that state-of-the-art tools can solve in an intra-app analysis.

3 ApkCombiner

We now discuss the design and implementation of `ApkCombiner`. First we present an overview of the approach in Section 3.1 before providing details on how we address the case of conflicting code, typically same-name classes, when combining apps (cf. Section 3.2). Although, for the sake of simplicity, we describe the case of merging two apps, the approach, and the prototype tool, can perform on merging any number of apps.

3.1 Overview

The main objective of our work is to enable Android-targeted state-of-the-art static analysis tools, which have proven to be effective in *intra*-app analyses, to perform as well in *inter*-app analyses. `ApkCombiner` takes a set of Android apps as input and yields a new Android app in output. The newly generated app contains all the features of the input apps except for their IAC features: there is no more IAC but only ICC in the new generated app.

The different steps of how `ApkCombiner` works are shown in Fig. 3. Each app is first disassembled into *smali* files and a Manifest file using a tool for reverse engineering Android apk files, namely android-apktool [8]. Second, all files from the apps are checked together for conflicts and integrated (with conflicts solved) into a directory. The Manifest files, one from each app, are merged into a single Manifest file. Finally, `ApkCombiner` assembles the smali files and the Manifest file along with all other resources, such as image files, into a single apk. Although

[8] https://code.google.com/p/android-apktool/

potential conflicts on such extra-resources may be met, `ApkCombiner` does not take them into account since the objective is not to produce a runnable apk, but an apk that can be analyzed statically.

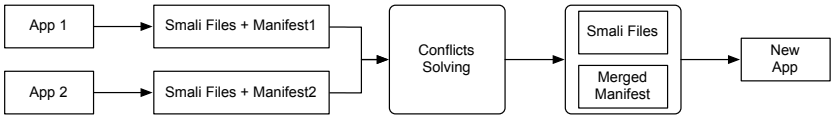


Fig. 3: Working steps of `ApkCombiner`

3.2 Resolution of Conflicts

Our prototype of `ApkCombiner` is focused on solving conflicts that may arise in the merging of code from two different apps. Such conflicts occur when two classes have the same name (up to the package level, i.e., the absolutely full qualified name). Thus, given class c_1 in app a_1 and class c_2 in app a_2, if $name(c_1) = name(c_2)$, we consider that there is a conflict between a_1 and a_2.

Fig. 4 illustrates the process of conflict checks we use. `ApkCombiner` considers that there is no conflict when two classes are named differently. If the name of two classes are the same, `ApkCombiner` distinguishes two cases according to the content of the classes. In a first type of conflict, the classes share the same name and their content is also the same (after verification of their footprint with the cryptographic hash), In this case, one copy of the class files is simply dropped. In the second type of conflict, i.e., when the content of the conflicting files are different, a thorough refactoring is necessary. This type of conflict occurs when, for example, two classes are actually from two different versions of the same library used in the two apps.

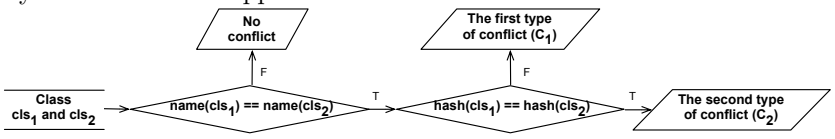


Fig. 4: The conflict checking process of `ApkCombiner`. Class cls_1 and cls_2 are from different apps.

Algorithm 1 details the described strategy for solving conflicts during merging as implemented by the procedure `CheckAndSolveConflicts()`. Given two sets ($set1$ and $set2$) of class files corresponding to the code of two apps (a_1 and a_2), the algorithm must identify and manage all conflicts.

First, two maps, referred to as $confliSameMap$ and $confliDiffMap$ are created to keep track of the classes that belong to the two types of conflict (lines 2-3). After identifying the kind of conflict that exists for each pair of classes across the two sets, the algorithm can attempt to solve the eventual conflicts. This resolution is performed in a two-step process. In step 1 (lines 17-22), the algorithm addresses the cases of type 1 conflicts. In step 2, type 2 conflicts are solved by refactoring the code.

Refactoring the code to solve conflicts is not as straightforward as renaming the conflicting classes. Indeed, there is a lot of dependencies to consider within

Algorithm 1 Checking and solving conflicts

```
1: procedure CheckAndSolveConflicts(set1, set2)
2:     confliSameMap ← new Map()
3:     confliDiffMap ← new Map()
4:     for all cls1 ∈ set1 do
5:         if set2.contain(cls1) then
6:             cls2 ← set2.get(cls1)
7:             if hash(class(cls1)) == hash(class(cls2)) then
8:                 confliSameMap.put(cls1, cls2)
9:             else
10:                confliDiffMap.put(cls1, cls2)
11:            end if
12:        end if
13:    end for
14:    if empty(confliSameMap, confliDiffMap) then
15:        return
16:    end if
17:    for all cls1, cls2 ∈ confliSameMap do
18:        remove class(cls2)
19:        if isComponent(cls2) then
20:            remove cls2 from Manifest2
21:        end if
22:    end for
23:    for all cls1, cls2 ∈ confliDiffMap do
24:        rename cls2
25:        solvingDependence(cls2, set2)
26:        if isComponent(cls2) then
27:            rename cls2 in Manifest2
28:        end if
29:    end for
30: end procedure
```

the code of other classes. Procedure `solvingDependence()`, in line 25, is used to handle these dependencies, where we take into account three types of dependencies: 1) for a given class c we need to rename, another class c_i may use it as one of its attribute, 2) method m_i of class c_i may hold a parameter of c and 3) statement s_i of method m_i may use c as a variable. For the third type of dependency, we deal with statements that instantiate the variable as well as access the variable's attributes and methods because only such statements hold information related to class c.

To combine multiple Android apps to one, we need not only to integrate the different apps' bytecode, but also to merge their Manifest files. In particular the merge of Manifest files must take into account the fact that some classes where dropped while others were renamed. If those classes represent Android components, and not helper code, these changes should be reflected in the final Manifest of the new app (line 20 and 27).

4 Evaluation

To assess the efficiency of our approach, we must evaluate the run time performance of `ApkCombiner` to ensure that this does not hinder its practical usability (cf. Section 4.1). Then, using a dataset with real-world apps, we check whether our approach is, in the end, capable of enabling state-of-the-art intra-app analyzers to support *inter-app* analysis (cf. Section 4.2).

Hypotheses. To run our experiments, we start with the assumption that the inter-app analysis may reveal significant security issues when a malicious application can exploit a leak in another, or when two apps can collude to leak data. To that end, we select a dataset containing both benign and malicious apps, and assume that, by pairs, they may cohabit in the same device.

Experimental Setup. We select two app sets G and M for our evaluation, where G is a set of apps randomly selected from Google Play store and M is a set of malicious apps. These malicious apps were recognized as such after analysis by VirusTotal antivirus products: we consider that an app is "really" malicious when at least 20 different antivirus flag it as such. Both G and M consist each of 3,000 Android apps. Then, for each app g_i randomly selected from G, we associate an app m_i, also randomly selected from M. This random combination only considers the possibility that two apps in one device may be independently installed by a user on his device. This kind of association provides 3000 opportunities of merging to `ApkCombiner`.

Our prototype tool succeeded in combining $2,648$ (88.3%) pairs of apps. Most failures were actually due to the limitations of android-apktool[9]. A few failures must however be attributed to the current implementation strategy of the refactoring process in `ApkCombiner`. These failures however are currently under investigation to improve the tool.

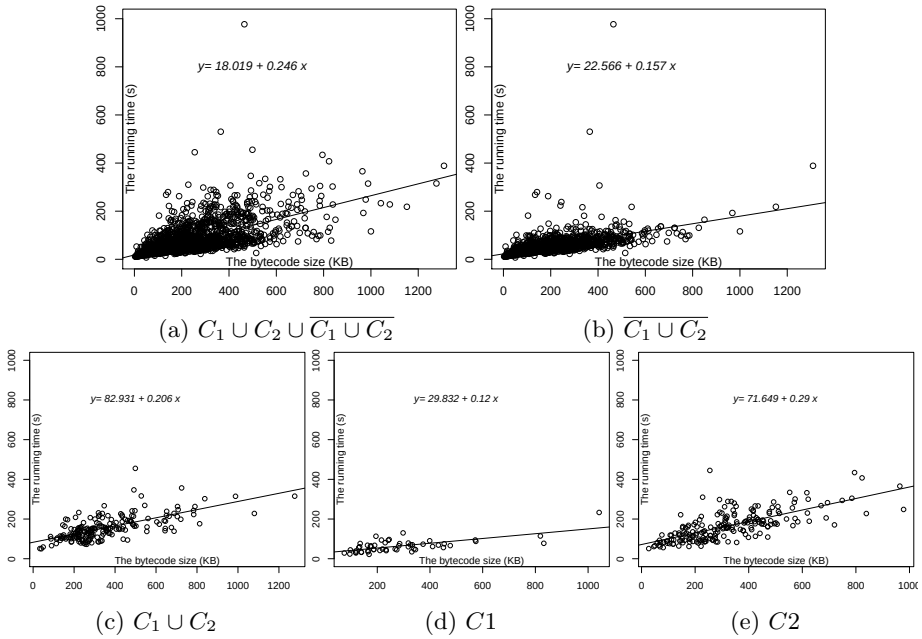
During the process of successful combinations, `ApkCombiner` solved 1,789 first type conflicts in 322 (12.2%) cases of combining pairs of apps. `ApkCombiner` also addressed 3,557 second type conflicts in 493 (18.6%) combination cases.

4.1 Time performance

The evaluation of time performance investigates the scalability of our approach. Indeed, a user may have on its device dozens apps that cohabit together. Thus, the inter-app analysis may require a fast combination of all those apps. Fig. 5 plots the running times[10] of `ApkCombiner` for each combined app. The running time is plotted against the sum size of each pair of apps (we use the bytecode size, as resource files that are not considered in the merging may introduce a bias).

[9] android-apktool often throws `brut.common.BrutException` for some combinations (e.g., *could not exec command* or *Too many open files*).

[10] Note that in this paper we consider the wall clock time (from start to finish of the execution). That means not only the actual CPU time but also the waiting time (e.g., waiting for I/O) are taken into account.



(a) $C_1 \cup C_2 \cup \overline{C_1 \cup C_2}$ (b) $\overline{C_1 \cup C_2}$

(c) $C_1 \cup C_2$ (d) $C1$ (e) $C2$

Fig. 5: Time performance against the byte code size. C_1 represent the set of combinations where first type conflicts were solved, while C_2 represents the set of combinations with second type conflicts.

Let C_1 and C_2 represent the successful combinations where conflicts of, respectively, first type and second type were solved. Consequently, $C_1 \cup C_2 \cup \overline{C_1 \cup C_2}$ represents all the successful combinations.

Fig. 5a plots the time performance for all combinations. The linear regression between the plots shows that there is a correlation between the execution time and the bytecode size. Comparing with Fig. 5b, we note that the slope of the regression is lower when we do not consider combinations that lead to conflicts. The limited difference in slope values (0.246 against 0.157) indicates that the conflict solving module is not a runtime bottleneck.

The differences between Fig. 5c, Fig. 5d and Fig. 5e further confirm how the resolution of second type conflicts requires more execution time than the resolution of first type conflicts.

4.2 Inter-app analysis

We consider IccTA [16], a state-of-the-art Android intra-app analysis tool, which originally aims at detecting inter-component privacy leaks inside a single Android app. We select IccTA to validate our approach by investigating the effectiveness of `ApkCombiner` in supporting existing tools for performing inter-app analyses.

With `ApkCombiner` we build app packages by combining pairs of apps. We then feed IccTA with these newly generated apps and assess its analysis results. We evaluate the use of IccTA in combination with `ApkCombiner` in two steps.

In the first step, we evaluate the impact of `ApkCombiner` on DroidBench, which includes three test cases related to inter-app communication leaks. We found that IccTA is able to report inter-app privacy leaks for the analyzed apps by analyzing the combined package provided by `ApkCombiner`. To the best of our knowledge, DidFail is currently the only tool which claims to be able to perform static inter-app analysis for privacy leaks. We therefore compare DidFail with our approach associated to an existing state-of-the-art tool for intra-app analysis. The results in Table 3 based the DroidBench benchmark show that IccTA, while it cannot handle inter-app analysis alone, outperforms DidFail when it is supported by `ApkCombiner`. The reason why DidFail fails on two test cases is that at the moment DidFail only focuses on Activity-based privacy leaks.

Table 3: Comparison between IccTA, DidFail and `ApkCombiner`+IccTA.

Test Case (from DroidBench)	IccTA	DidFail	`ApkCombiner`+IccTA
InterAppCommunication_startactivity1	✗	✓	✓
InterAppCommunication_startservice1	✗	✗	✓
InterAppCommunication_sendbroadcast1	✗	✗	✓

In the second step, we evaluate `ApkCombiner` on 3,000 real Android apps. We first build an IAC graph through the results of our extended Epicc [16, 18], where an app stands for a node and an inter-app communication is modeled as an edge. For each of such edges, we launched `ApkCombiner` on the associated pair of apps and then used IccTA on the generated app.

We were thus able to discover an IAC leak between app *Ibadah Evaluation*[11] and app *ClipStore*[12]. In the Ibadah Evaluation apk code, the source method *findViewById* is called in component `com.bi.mutabaah.id.activity.Statistic`, where the data of a *TextView* is obtained. Then this data is stored into an Intent along with two extras, a subject named `android.intent.extra.SUBJECT` and the text referred to as `android.intent.extra.TEXT`. Subsequently, the triggering method *startActivity* is used to transfer the Intent data to the ClipStrore app which extracts the data from the Intent with the same extra names and writes all the data into a file named `clip.txt`. Note that we consider saving sensitive data into disk as a leak.

5 Discussion

Scalability. As introduced in Section 4.1, `ApkCombiner` scales linearly with the bytecode size. Unfortunately, in practice, when increasing the bytecode size (e.g., increasing the number of apps to combine), the processing time and memory requirement of Android analysis tools (e.g., IccTA or FlowDroid) also grow significantly. Thus such approaches may not be scalable when running on top of `ApkCombiner`. To limit the impact of this scalability issue, a possible approach is to limit the number of Android apps to combine. This is a reasonable limitation, as the number of apps obstructs the work of attackers as well. For example, given

[11] https://worldapks.com/ibadah-evaluation/, `com.bi.mutabaah.id` in our dataset.
[12] https://worldapks.com/clipstore/, `jp.benishouga.clipstore` in our dataset.

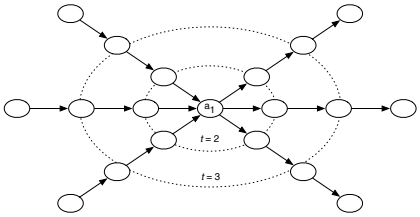


Fig. 6: An example of an IAC graph and a trade-off threshold t.

an effective attack, the more number of apps involved, the more complexity of building such attack introduced and the less probability of a user installing all of the involved apps. Our solution is to build an IAC graph to represent the dependencies among apps, the idea being that if there is no link (edge) between two apps (nodes) there is no need to combine them. Based on the IAC graph, we introduce a threshold t to denote the maximum number of apps `ApkCombiner` may combine together. The trade-off limitation length t enables existing intra-app analyzers to remain scalable when used with `ApkCombiner`.

Let us take Fig. 6 as an example, which shows an IAC graph and the concept of threshold t. For app a_1, if we set $t = 2$, then we only need to run `ApkCombiner` 6 times (the small circle) and most importantly we only need to combine 2 (or t) apps each time. Notice that with the built IAC graph, new apps can be added to the graph in an iterative and incremental manner. When new apps are involved, we only need to add them to the existing IAC graph. We do not need to run the previously computed apps again when adding the new apps. In short, by building an IAC graph and setting up a threshold t, the original set of Android apps is split into multiple small sets that both `ApkCombiner` and the state-of-the-art intra-app analysis tools can analyze.

Limitations. At the moment, we do not offer a guarantee that the newly generated app can be executed. Except from the bytecode and Manifest, we simply combine all the other resources such as native code, layout files without checking whether they are conflicted or not. This may result in errors for analysis tools that rely on such resources.

6 Related Work

To the best of our knowledge, in the Android community, our approach is the first work that attempts to complement existing state-of-the-art intra-app analysis tools to indirectly support inter-app analyses. Our approach is also the first proposal that supports context-aware inter-app analysis. However, researches on detecting IAC vulnerabilities are not new.

Privilege escalation attack, an IAC vulnerability, has been studied by a large body of works [3, 6, 8]. Davi et al. [6] show that a genuine app can be exploited at runtime and a malicious app can escalate granted permissions. Prominent examples of privilege escalation attacks are *confused deputy* and *collusion attacks* [3]. Confused deputy attack is about the possibility for malicious app to exploit another privileged (but confused) app's vulnerable interface. Collusion

attack concerns the collusion of apps that combine their permissions to be able to perform actions beyond their individual privileges. Our approach differs from theirs because we are focusing on supporting static inter-app analysis, while they are using dynamic testing to detect such vulnerabilities.

ComDroid [4] analyzed inter-app communication in Android apps and discovered IAC vulnerabilities such as *Broadcast Injection* and *Activity Hijacking*. Epicc [18] is another tool that dedicated to identify IAC vulnerabilities in Android apps. Besides, Epicc records the actual values of IAC objects, which makes it appropriate to build inter-component (or inter-app) links. ContentScope [22] is another tool which detects `Content Provider` based vulnerabilities. It argues that a `Content Provider` component can leak sensitive data to other apps and malicious apps can also pollute data maintained by a `Content Provider`. More recently, PCLeaks [17] was proposed to perform data-flow analysis on the above IAC vulnerabilities to discover potential component leaks, which may leak private data across Android apps. While the above static approaches are tackling IAC vulnerabilities, they are actually only analyzing one app at a time and their outputs are so-called potential results. Our approach is able to complement them by enabling them to indirectly perform inter-app analysis and give them an opportunity to conform that the aforementioned potential vulnerabilities are exploitable in real-world apps.

To the best of our knowledge, there is only one, recent, static approach, DidFail [13], which is able to perform static inter-app analysis. However, as shown in Section 2.2 (type (B_1)), DidFail simply combines the results of intra-app analyses following an approach which is neither context-aware nor general. In contrast, our approach is able to provide a general context-aware inter-app analysis, and therefore, all intra-app analyzers can benefit from it.

7 Conclusion

We discussed `ApkCombiner`, a tool-based approach for reducing an Inter-App Communication problem into an intra-app Inter-Component Communication problem by combining multiple Android apps into one. After the combination, existing intra-app analysis approaches can be applied on the generated Android app to indirectly report inter-app results. Since we combine apps at code level, our approach is context-aware and general. We evaluate `ApkCombiner` to demonstrate that, despite a conflict resolution algorithm that requires a time-consuming refactoring process, the approach is scalable. We further showed that it can improve the capabilities of existing state-of-the-art tools. For example, we showed that using `ApkCombiner` can enable tools such as IccTA to discover IAC privacy leaks in real-world apps.

Acknowledgments. This work was supported by the Fonds National de la Recherche (FNR), Luxembourg, under the project AndroMap C13/IS/5921289, by the BMBF within EC SPRIDE, by the Hessian LOEWE excellence initiative within CASED and by the DFGs Priority Program 1496 Reliably Secure Software Systems and the project INTERFLOW.

References

1. K. Allix, Q. Jerome, T. F. Bissyande, J. Klein, R. State, and Y. L. Traon. A forensic analysis of android malware–how is malware written and how it could be detected? In *COMPSAC*. IEEE, 2014.
2. S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI 2014*, 2014.
3. S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *NDSS*, 2012.
4. E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *MobiSys*, New York, NY, USA, 2011. ACM.
5. X. Cui, D. Yu, P. Chan, L. C. Hui, S. Yiu, and S. Qing. Cochecker: Detecting capability and sensitive data leaks from component chains in android. In *ACISP'14*.
6. L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *Information Security*, pages 346–360. Springer, 2011.
7. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.
8. W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.
9. W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *USENIX Security*, 2011.
10. A. P. Fuchs, A. Chaudhuri, and J. S. Foster. Scandroid: Automated security certification of android applications. *Manuscript, Univ. of Maryland*, 2009.
11. J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *NDSS*, 2004.
12. M. Haris, H. Haddadi, and P. Hui. Privacy leakage in mobile computing: Tools, methods, and characteristics. *arXiv preprint arXiv:1410.4978*, 2014.
13. W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. Android taint flow analysis for app sets. In *SOAP@PLDI*, pages 1–6. ACM, 2014.
14. M. La Polla, F. Martinelli, and D. Sgandurra. A survey on security for mobile devices. *Communications Surveys & Tutorials, IEEE*, 15(1):446–471.
15. P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The soot framework for java program analysis: a retrospective. In *CETUS*, 2011.
16. 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 *ICSE*, 2015.
17. L. Li, A. Bartel, J. Klein, and Y. Le Traon. Automatically exploiting potential component leaks in android applications. In *TrustCom*. IEEE, 2014.
18. D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *USENIX Security*, 2013.
19. D. Wagner and D. Dean. Intrusion detection via static analysis. In *S&P*, 2001.
20. F. Wei, S. Roy, X. Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *CCS*, 2014.
21. L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The impact of vendor customizations on android security. In *CCS*, pages 623–634. ACM, 2013.
22. Y. Zhou and X. Jiang. Detecting passive content leaks and pollution in android applications. In *NDSS*, 2013.