J Comput Virol Hack Tech
DOI 10.1007/s11416-017-0296-4

ORIGINAL PAPER

Automated generation of colluding apps for experimental research

Jorge Blasco[1] · **Thomas M. Chen**[2]

Received: 17 October 2016 / Accepted: 20 March 2017

Abstract Colluding apps bypass the security measures enforced by sandboxed operating systems such as Android. App collusion can be a real threat in cloud environments as well. Research in detecting and protecting against app collusion requires a variety of colluding apps for experimentation. Presently the number of (real or manually crafted) apps available to researchers is very limited. In this paper we propose a system called Application Collusion Engine (ACE) to automatically generate combinations of colluding and non-colluding Android apps to help researchers fairly evaluate different collusion detection and protection methods. Our initial implementation includes a variety of components that enable the system to create more than 5,000 different colluding and non-colluding app sets. ACE can be extended with more functional components to create even more colluding apps. To show the usefulness of our system, we have applied different risk evaluation and collusion detection methods to the created set of colluding apps.

Keywords Collusion · Android · Malware · Sandbox · Benchmark

✉ Jorge Blasco
Jorge.BlascoAlis@rhul.ac.uk

Thomas M. Chen
Tom.Chen.1@city.ac.uk

[1] Present Address: Information Security Group, Royal Holloway, University of London, Egham, UK

[2] Department of Electrical and Electronic Engineering, City, University of London, London, UK

1 Introduction

Modern mobile operating systems, such as Android, use sandboxing to prevent malicious apps from causing harmful effects by restricting each process from accessing resources outside its domain. In a sandboxed environment, access to sensitive system resources is mediated by the operating system and restricted by default. Apps requiring access must request the necessary permissions from the user at installation or execution time. Additionally, resources from other apps are outside of the boundaries of the sandbox and must be accessed through inter-application communication methods, if available in the operating system.

Colluding apps use covert and overt channels to jointly perform malicious operations [7]. The origin of app collusion can be traced back to the *confused deputy* attack [23]. Confused deputies expose protected resources through public interfaces. In Android, confused deputy attacks can happen in the form of *permission re-delegation attacks* [15,19,33]. A careless developer may unintentionally expose permission-protected resources by allowing the component that access those resources to communicate with other apps through IAC (inter-app communication). An attacker can take advantage of this component to access the protected resource without requesting the corresponding permission.

Colluding apps behave similarly to malicious apps taking advantage of confused deputies, but their actions are executed on purpose. Colluding apps can carry out information theft attacks but also can be used to misuse a device service or increase the impact of an attack inside a system. The main goal of collusion attacks is to avoid the restrictions imposed by sandboxed environments, like Android, to make an attack more difficult to detect. Users, security researchers and malware analysis services normally focus on the the access to resources given to an app to establish its risk level. If that access is split across several apps that collude, an app may not require to request access to a protected resource. It will only have to ask for it to a colluding app that already has access to it. It must be noted that, while Android requires the user to grant the permissions to be used by an app, it does not impose any control on how apps exchange information. Although app collusion is not a widespread problem today, there have been already some cases of malicious apps engaging in colluding behaviors. In [6] authors detected some samples from the VirusShare project [1] that were receiving data through broadcast receivers and sending it through SMS messages. More recently, researchers discovered a malicious version of the MoPlus SDK that was synchronizing the execution of the malicious payload through app collusion [2,9]. All apps running on a device embedding the MoPlus SDK would talk to each other to determine which of these apps had the most privileges. The app with the most privileges would be the only one executing a local HTTP server to receive commands from a command and control server. This SDK was embedded in more than 5,000 versions of 20 apps. This synchronization strategy was used by the developers of the malicious SDK to avoid apps embedding the SDK but with not enough privileges to activate the malicious payload. In this way, only the payload within the app with the required permissions would execute, maximising the result of the attack.

Malware researchers have access to public datasets, predominantly for Android, that can be used to test their detection methods [4,35]. This allows fair evaluation and comparison between proposed methods, which in the end fosters better quality research. Unfortunately, representative datasets do not presently exist for colluding apps because the very few examples of collusion happening in the wild, have been discovered very recently.

This paper aims to meet the need a practical set of colluding apps for research. Our system called *Collusion Application Engine* (ACE) is capable of automatically generating multiple colluding app sets with a variety depending on the configuration of app component templates and code snippets. ACE can be extended with new app components and code blocks to create a greater variety of new colluding app sets, if needed. In this way, it is easy to create substantial app sets (colluding and non colluding) for experimentation avoiding the need for a great deal of manual programming effort. The source code of ACE, as well as an initial set of 240 apps are available upon request from the authors.[1]

The remainder of this paper is structured as follows. Section 2 describes previous efforts of other researchers in exploring colluding apps. Section 3 describes the methodology underlying ACE and how it can generate thousands of different app sets. Section 4 shows the validation process followed to test the apps created by ACE. Finally, Sect. 5 presents our conclusions and future directions for research.

2 Related work

Android malware detection has been an attractive and active research area during the last few years. As a result, techniques for detecting Android malware are readily available [17,30]. These can be categorised into two main groups: static and dynamic. In static analysis, certain features of the app binary are extracted and analysed using different approaches such as machine learning techniques. Examples of these are [5], using hardware components, requested permissions, critical and suspicious API calls, and network addresses or many others [11,14]. Conversely, dynamic analysis detects malware at run-time. It deploys suitable monitors on Android systems to log traces and features that are used to look for malicious behaviours. Examples of these are [21], which keeps track of the network traffic or [24], which collects information about the usage of network usage, memory and CPU.

In contrast to malware detection, detecting colluding apps involves not only obtaining features that show if an app carries out a security threat, but also revealing whether communication between several apps occurs during the attack. As most of the existing malware detection techniques focus only on detecting whether single apps can carry the full extent of the threat and not on their communication channels, they are naturally constrained to detect collusion. Taint analysis based approaches like Amandroid [32] and FlowDroid [6] could be used for collusion detection. These are focused on analyzing single apps to detect information leaks through inter-component communications, ICC, (i.e. a location leaking from a service to an activity within the same app). This limits its usefulness against colluding apps. First, they are only able to analyse single apps. This means that, although they are able to detect leaks to other apps through inter-app communications,[2] (i.e. and activity/service from one app sending information to an activity/service from another app), they are not able to tell the other app that is taking part in the collusion. In addition to this, colluding apps may use other communication channels for collusion (i.e. covert channels) rather than standard IAC channels.

To overcome this limitation, there are approaches like APKCombiner [25] that join two applications into a single APK. This enables information flow tools to analyze app pairs.

The first known example of app collusion is a proof-of-concept app named *Soundcomber* [29]. The first app, which requires only access to the device microphone

[1] The dataset is available on http://personal.rhul.ac.uk/udai/003/colluding_apps.zip.

[2] In Android, IAC and ICC are implemented through very similar APIs.

(RECORD_AUDIO permission), listens for calls to telephone banking services and extracts the digits pressed by the user. The second app, which requires only Internet access (INTERNET permission), transmits the stolen information to a remote server. Sensitive information extracted by the first app is transmitted to the second app using standard Android inter-application communications (IAC) and covert channels (file locks, settings modifications, etc.).

The other colluding apps available in the literature can be categorized into two groups: those developed to test detection and protection methods and those developed to explore the different covert channels available in Android.

A combination of Soundcomber, three proof-of-concept colluding app sets, and another three vulnerable apps from other works [15,18,26] were used to test an an operating system extension called XManDroid [10]. The colluding apps were capable of stealing user contacts, SMS messages, and location, respectively. In a similar way, [7] describes 10 colluding apps developed to evaluate their static analysis detection methods. In this case, colluding app sets are not restricted to information theft. One of the developed sets is also able to send premium-rate SMS messages to numbers that are received from another app. Finally, the authors of [8] evaluated their proposal against 13 colluding apps that steal sensitive information and communicate through intents and DroidBench [20]. DroidBench is the only public dataset that includes colluding apps. However, it only includes three colluding apps from 120, as it is intended for evaluating the effectiveness of taint-analysis tools (where collusion is a subset).

Overall, 36 different apps have been accounted as developed with the specific purpose of testing collusion detection methods. The development of mobile apps is a time consuming task that requires to gain knowledge of the app development environment and many hours of testing. An automated method to develop colluding apps could reduce the efforts researchers have to spend on these time consuming tasks, so they can focus on improving the actual detection methods. As an example of this, we use our automatic app generation method to generate 240 colluding app pairs that are tested with two different collusion detection methods.

Besides Soundcomber, several previous papers have investigated the usage of covert channels for app collusion. Covert channels in Android, as in other systems, are restricted by the amount of shared resources, side channels that can be found in the device, and human imagination [13,34]. [27] describes a collusion scenario where a *ContactManager* and *PasswordManager* app use overt and covert channels to extract information through another app that acts like a generic weather app. In [28], the authors enumerate and evaluate the bandwidth of different overt and covert channels in real devices. Covert channels tested include: intent type enumeration, settings modification, thread and socket enumeration and free disk space among others, which were used also in [22]. An imaginative covert channel that uses the device actuators and sensors is described in [3]. The vibration motor is used to transmit information while the accelerometer sensor is used to capture it from another app. More recently, [16] showed how repackaging can be used to inject colluding payloads into benign apps. Repackaged apps communicate through a covert channel based on process enumeration. ACE could be used to generate, with minimal effort, additional testing apps based on the specific features of these specifically developed colluding apps (covert channels and payloads). The rest of this sections describes in more detail each of the components of ACE.

3 Application collusion engine

ACE aims to fill the need for colluding app datasets for research experimentation. It can free up significant time for researchers so they can focus instead on efforts to develop collusion detection and protection methods. If a new covert channel or attack is found, ACE can be easily extended. In this section, we describe the system design and how it can generate colluding app sets.

3.1 General overview

Generating individual malicious apps is a relatively simple task. A malicious payload can be injected into a template or repackaged app, modifying the required permissions as needed. Generating colluding app sets is more complex because the creation of one app needs to take into account how the rest of the apps in the colluding set were generated.

ACE is composed by two main components: the *Colluding Set Engine* and the *Application Engine* (Fig. 1). In a nutshell, the Colluding Set Engine tells the Application Engine how it should create apps in order to collude. The Colluding Set Engine reads *collusion description files* from the *Collusion Template* database and, using the *App template* database generates a set of *application description files*. The Application Engine reads the app description files passed by the Collusion Set Engine; it fetches the necessary payloads from the *Code Snippet* and *Component Templates* databases and builds the app files, producing a signed *apk* file for each app of the colluding set.

3.2 Colluding set engine

The Colluding Set Engine tells the Application Engine how to generate apps in such a way that they end up colluding.

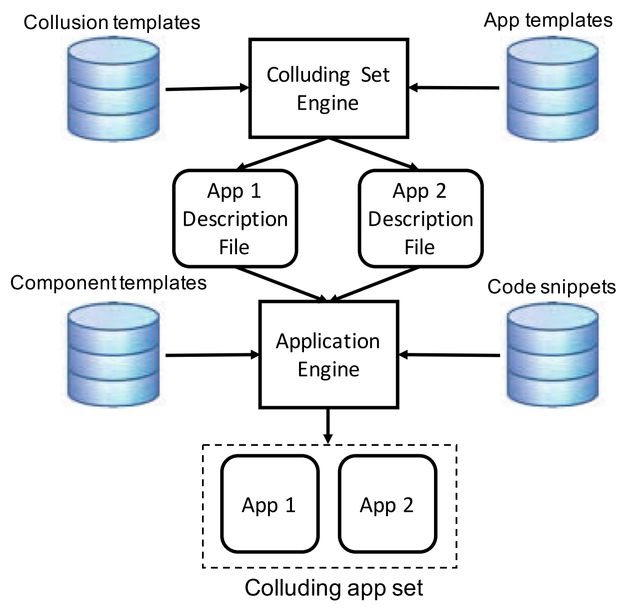

Fig. 1 General overview of the application collusion engine (ACE)

3.2.1 Collusion templates

A collusion template describes the different apps that take part in a colluding app set. This is, the threat they carry out and how they communicate. Colluding apps can carry out any attack similar to the ones posed by single apps [30]: denial of service, service misuse, and information and money theft. Although the nature of each of these attacks is different, they are all based on executing a set of actions in a specific order.

Our colluding templates follow this philosophy. For each of the aforementioned threats, we have established a set of actions that are required to execute it. For example, an information theft attack will require (i) reading sensitive information and (ii) sending it outside to a remote server. In the same way, a ransomware attack will require (i) encrypting the personal files (ii) processing or facilitating the payment and (iii) decrypting back the files.

In our case, each defined action is implemented with a specific code snippet. In this way, creating a colluding app set to execute a threat requires to split, across different apps, the different code snippets (actions) required to execute it. The communication channel used to execute the attack is not relevant to the attack goal. As long as the selected code snippets allow apps to exchange information somehow, the generated set will be able to collude. This fact enables us to create many different colluding app sets for even the same threat by interchanging different communication code snippets. Additionally, this allows us to create colluding app sets of an arbitrary number of apps. If the number of apps in the set exceeds the number of actions, some of the apps will just forward messages from one app to another.

3.2.2 App templates

An app template includes all the initial files, organized in an Android project structure, that are required by the Android Development Kit to build and sign an app. This includes, among other things, an initial manifest file, build and signing scripts, a private key to sign the resulting apk, and all the other resources that may be required during the process. App templates can be customised to target specific Android versions or include other specific features such as loading images, etc.

3.3 Application engine

The Application Engine creates fully working apps by filling app component templates with code snippets. The Application Engine works as follows. First, it reads and processes the app description file. This retrieves the required component templates and code snippets and adds all the reference resource files to the app source. Then, the component templates are filled; the app manifest is generated; and the project is built and signed, producing an APK file that can be installed in a device.

3.3.1 Component templates

Apps in Android are composed of four different components:

- *Activities* represent screens of the user interface. Activities allow the user to interact with the app, giving back some feedback. Activities run only on the foreground. Apps are generally composed of a set of activities.
- *Services* execute operations in the background. They are generally used by other components of the app to perform long-running tasks that must be executed in the background: listening to incoming connections, play music, download a file, and so on. Services can be configured so they can be accessed remotely by other processes.
- *Broadcast Receivers* respond to broadcast messages that can be sent through Intent objects, by the same or other apps in the device.
- *Content Providers* manage access of other apps to the app's own shared data. Apps with content providers enable other apps to read and write data inside their sandbox.

The Application Engine generates these components based on templates stored in the component templates database. Each app component can be defined by different templates, providing more variety across the generated apps. A component template describes the main structure of the component and injection points. Injection points allow to fill the template with the necessary information such as package name, imports, strings or code. For example, the activity shown in (Listing 1) has injection points for its name (`$name`), package (`$package`), layout file (`$layout_file`) and code snippets (`$code`). The value of these injection points is defined by the application description files generated by the Collusion Set Engine. In some cases, these will be assigned a string like `$name` in our example. In others the injection points will be replaced by a code snippet, like in the case of `$code`. A more detailed description of the structure of Application Description Files is given in Sect. 3.3.3.

```
package $package;

import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;

public class $name extends ActionBarActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.$layout_file);
        $code
    }

    ...
}
```

Listing 1 Activity template example. $package, $name, $layout_file and $code are injection points

3.3.2 Code snippets

Code snippets are small portions of code that execute a specific function. These are used to populate the app components at code injection points (marked with `$code`). There are three kinds of code snippets, depending on the kind of function they enable to execute: resource, communication, and general snippets. A code snippet can also have *injection points*. These are used to specify parameters such as variable names, string names or other information relevant to the specific code. The app description files specify how the code snippets are injected into the component. The Colluding Set Engine is in charge of structuring them in a way such that the resulting apps are able to communicate and collude.

Resource Snippets access the permission-protected resources of a device. They can be used to access sensitive information (contacts, accounts, etc.) or to execute a particular function (sending a SMS message, starting a task, etc.). The injection points of a resource snippet define parameters such as the input and output variable names, required variables such as the app context, etc. Resource snippets generally require the app to request additional permissions. These are defined along the snippet. Listing 2 shows a code snippet used to get account data. Lines 1-3 specify the additional imports required. Line 4 specifies that any app including this code snippet must add the `GET_ACCOUNTS` permission. The code snippet includes two injection points. `$context` holds the name of the app context variable. This is required to access the accounts service. `$data` stores all the account information read from the device, so other code snippets can use it.

```
1  import android.accounts.Account;
2  import android.accounts.AccountManager;
3  import android.content.Context;
4  //permission:android.permission.GET_ACCOUNTS
5  AccountManager am = (AccountManager)
           $context.getSystemService(Context.ACCOUNT_SERVICE);
6  Account[] accounts_acs = accounts_am.getAccounts();
7  StringBuffer accounts_sb = new StringBuffer();
8  for(Account a : accounts_acs) {
9      accounts_sb.append(a.toString()+";");
10 }
11 String $data = accounts_sb.toString();
```

Listing 2 Code snippet used to access the accounts being stored in the device

Communication Snippets are in charge of executing communication tasks. There are two types of communication snippets: those used to communicate outside the device and those used for inter-app communication. Listing 3 shows a code snippet that posts a string to a URL. The snipppet creates a new threat to open an HTTP client and execute the post query. Lines 1-3 specify the required imports. Line 4 specifies that any app including this snippet should request the `INTERNET` permission. Lines 5-17 include the code required to post the message to the URL specified by the injection point `$url`. The injection point `$data` specifies the variable name or string value that should be sent to the remote server.

```
1  import org.apache.∗;
2  import java.util.ArrayList;
3  import java.util.List;
4  //permission:android.permission.INTERNET
5  final String toSendOutside = $data;
6  new Thread(new Runnable() {
7      public void run() {
8          HttpClient httpclient = new DefaultHttpClient();
9          HttpPost httppost = new HttpPost($url);
10         try {
11             List<NameValuePair> nameValuePairs = new
                       ArrayList<NameValuePair>(1);
12             nameValuePairs.add(new
                       BasicNameValuePair($postkey,
                       toSendOutside));
13             httppost.setEntity(new
                       UrlEncodedFormEntity(nameValuePairs));
14             HttpResponse response = httpclient.execute(httppost);
15             } catch (Exception e) {
16                 ...
17             }}}).start();
```

Listing 3 Code snippet used to post a string value to the internet

Code snippets also enable inter-component and inter-app communication. If the snippet sends information to another component of the app (i.e. an intent that launches a service within the same app) it will allow inter-component communication. In this case, the app will also have to include the definition of the component that will receive the communication and a code snippet to process that information. If the snippet sends information to a component that belongs to another app (i.e. a broadcast intent) it will allow inter-component communication. Of course, an app can include both ICC and IAC snippets (depending on the application description file used to generated them).

Communication snippets are generally used in pairs: a code snippet that sends information through a specific channel is used in one component and the code snippet that receives the information through that same channel is used in another component. If the snippets are placed within the same app they will enable inter-component communication. If the snippets are included in two different apps, they will enable inter-app communication. As an example, Listing 4 shows the code required to send information to other apps through a broadcast intent. This code snippet includes 4 injection points: `$intentaction` specifies the action that will be assigned to the intent; `$key` is a string value used as a key for the data that is being sent through the intent; `$data` specifies the variable name or literal value of the information to be sent through the intent; finally, `$context` requires the value of the variable that holds a reference to the app context (activity or service), which is required to send a broadcast intent.

```java
import android.util.Log;
import android.content.Intent;
Intent i = new Intent();
i.setAction($intentaction);
i.putExtra($key,$data);
$context.sendBroadcast(i);
```

Listing 4 Code snippet used to sent a string value to other application through an intent

Listing 5 shows the code snippet required to receive information sent by the previous intent-based code snippet. This code snippet declares, implements and registers a broadcast receiver. This could also be implemented by using a BroadcastReceiver app component template. `$intentaction` specifies the intent action that will be used to match the broadcast intent. `$key` stores the key where the transmitted data is being stored. `$data` will store the value extracted from the intent. Finally, `$code` is another code snippet that specifies the action to be executed with the data. To create a communication channel, the values of `$intentaction` and `$key` should match in both snippets.

```java
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
IntentFilter ifilter = new IntentFilter($intentaction);
this.registerReceiver(new BroadcastReceiver() {
    @Override
        public void onReceive(Context context, Intent intent) {
            String $data = intent.getStringExtra($key);
            $code
        }
    }, ifilter);
```

Listing 5 Code snippet used to receive a broadcast intent

In some cases, there might be synchronization issues. For example, if an app sends an intent and no app is registered to receive it, the information sent will be lost. In other cases, like when using external storage this problem does not happen. However, controlling this is not part of ACE, as execution order (or even installation) is controlled by the developer. When creating a pair of colluding apps, the Colluding Set Engine generates the app description files avoiding this, in such a way that the both apps can communicate.

General Snippets provide the general functionality to complete a fully working app. These include: string concatenation, encryption, variable initialization, logging, etc. These can be used to combine, for example, execution of an app that extracts information from the device and another that encrypts it before sending it through an inter-app communication channel.

Listing 6 shows a code snippet used for logging. This code snippet could also be used as a sending communication snippet in Android versions below 4.3.

```java
import android.util.Log;
Log.v($logtag,$tolog);
```

Listing 6 Logging code snippet

3.3.3 App description files

App Description Files specify which app components will be used within an app, and how their injection points will be filled. In some cases, injection points will be replaced with simple strings, while in others, they will be replaced with code snippets (`$code` injection points). Listing 7 shows the app description file of a very simple app composed by only one activity. The main element of an app description file is the `project` element. A project holds the rest of the app components and defines the package name of the app. The activity defined in this description file takes its template code from `resources/activities/SimpleActivity.java` and includes only one code block (inside the `onCreate` method. This example code snippet has three injection points. The first one initializes a string variable, `text`. The second registers a broadcast receiver that stores the value received through an intent into `text`. The last one logs the value of `text` with a specific tag (`VALUE`). The activity attribute `main` specifies if the activity is the main activity of the app.

```xml
<?xml version="1.0" encoding="utf−8"?>
<project template="resources/simple_app"
        package="com.acid.app" label="Receiver">
  <activity path="resources/activities" label="Receiver"
          name="SimpleActivity.java"
          layout="activity_simple.xml" package="com.acid.app"
          main="true">
    <code id="code">
      <codesnippet path="basic" id="string_init">
        <param id="var">text</param>
      </codesnippet>
      <codesnippet path="interapp_communications/intent"
            id="recv_dynamic_receiver">
        <param id="intentaction">"action.SEND"</param>
        <param id="data">text</param>
        <param id="key">"datakey"</param>
        <param id="code">
          <codesnippet path="basic" id="basic_log">
            <param id="logtag">"VALUE"</param>
            <param id="tolog">text</param>
          </codesnippet></code></codesnippet></code>
    </activity>
</project>
```

Listing 7 Example app description file

3.4 Building app sets

ACE can be used to create sets of colluding and non-colluding apps. The process of creating a colluding app set consists of the following:

- Read the collusion template that specifies the actions to be executed in each colluding app.
- Generate *n* apps where *n* is more than 1 and smaller than the number of actions in the template; assign at least one action to each app.
- Determine if there is going to be forwarding apps; if so, create them.
- Add pairs of communication snippets to enable communications between the apps in the set.
- Generate the app description files for each app in the set.
- Call the Application Engine to create the set.

Figure 2 shows an example of this process for a colluding app set that extracts the contact list from the user device. The set is composed of three apps. The first app reads the contacts and sends them to the second app via an intent. The second app forwards the information to the third app using external storage. The third app of the set sends the information to a remote server.

3.5 Generating non-colluding apps

ACE also allows the easy generation of non-colluding apps. This can be achieved by simply creating an app description file (Fig. 2) that includes the necessary components and code snippets for the required functionality. App description files are automatically generated by the Collusion Set Engine for colluding app sets, but can be manually created and used directly with the Application Engine. This fact can be used to generate apps that might be detected as colluding but are not colluding. These generated apps could be used to test, against false positives, new collusion detection methods. To do this, we have followed these approaches:

- Create app sets that exchange information, but do not access sensitive resources.

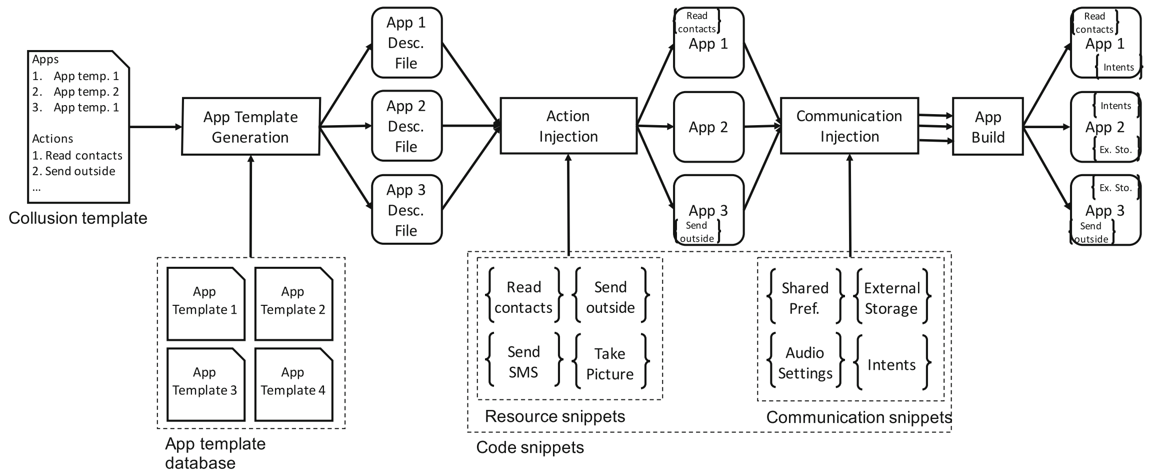

Fig. 2 Detailed example of all the processes executed by ACE

Table 1 Resource code snippets used for the generation of colluding app sets executing information theft attacks

Snippet	Description
Accounts	Returns device accounts information
Bookmarks	Returns the browser bookmarks
Call log	Returns the call log
Contacts	Returns the list of contacts
History	Returns the navigation history
IMEI	Returns the device IMEI
Microphone	Records audio for 5 seconds
WiFi	List of WiFi SSID networks
Tasks	List of processes being executed

– Create app sets that access sensitive resources but exchange information that is not related to those sensitive resources.
– Create apps that access sensitive resources and send information to non-colluding apps via standard Android communication channels (e.g. the generated app reads your location and allows sharing it via facebook).

4 Experiments

The amount of colluding app sets that can be generated with ACE depends greatly on the number of available code snippets and templates. We have validated our tool by generating a set of colluding apps and measuring its risk with different methodologies available in the literature. This section describes this process.

4.1 Colluding app set generation

We generated 240 pairs of colluding apps focusing on apps that execute information theft attacks because most of the detection tools are focused on that threat. Table 1 lists the different resource snippet used for app generation. Each of the 240 generated app sets includes one app that reads sensitive information from the device. The code that reads the sensitive information in that app is created by randomly injecting a combination of the 9 snippets shown in Table 1. All the information read by the snippets is concatenated into an string and then sent to the other app for extraction.

In our experiments we have focused on two different communication channels:

– Intents: an app launches a broadcast intent with a randomly generated action. The receiving app registers a broadcast receiver with the same action. Other intent based communication channels (e.g. explicit intents launching activities and services) could be easily added by incorporating the corresponding code snippets.
– Shared Preferences: are an Android feature that allows apps to store key-value pairs of data. configuration and preferences. Although it is not intended for inter-app communication, apps can use key-value pairs to exchange information if proper flags are defined (`WORLD_READABLE` or `WORLD_WRITABLE`) when accessing and storing data. In our sets, one of the apps saves the data into a world readable shared preference file. The receiving app accesses the same file to read the information.

Each communication channel has been used in half of the app sets. Thus, overall there are 120 app sets (240 apps) that execute information theft attacks using intents and other 120 app sets (240 apps) that execute information theft using shared preferences as communication channel.

4.2 Validation of collusion behavior

All the generated app sets have been tested on a real smartphone to verify that the collusion attack is realised when both apps are present in a device. This process was automated with a script, using *monkeyrunner*,[3] that installs, executes and verifies that the collusion attack happened on a real smartphone (Moto E with Android 4.3). Specifically, the script executed the following tasks for each pair of generated colluding apps:

– Install both apps.
– Run the first installed app for five seconds (in the foreground).
– Press the home button.
– Run the second app for five seconds (in the foreground).
– Press the home button.
– Uninstall both apps.

To validate that the sensitive information was exchanged between colluding apps, we manually created contacts, and personal information on the device prior the tests. In addition, information sent to a external web server (controlled by us) was logged on the server and the device. To ensure that the collusion really happened we verified the device logs and the HTTP POST requests received by the server.

4.3 Measuring collusion risk

Current App collusion detection techniques can be split in two groups: operating system extensions and taint analysis

[3] https://developer.android.com/studio/test/monkeyrunner/index.html.

tools. The first group focuses on detecting and mitigating collusion during execution while the second group focuses on analysing the static features of the app code and resources without executing the apps. Unfortunately, we were unable to find any working version of an operating system extension, like XManDroid, to execute our experiments.

FlowDroid and Amandroid are two examples of taint analysis proposals [6,32]. The focus of these tools is to detect sensitive information flows between components within the same application (ICC), but they are also capable of detecting, with certain limitations, when an app component leaks sensitive information to other apps through inter-app communication (IAC). Their limitation relies on the fact that the tools are only able to tell if sensitive information is being sent to other apps via IAC, without specifying the actual apps, as the analysis is only executed over one app. In order to identify if two apps communicate, the analysis must be performed over the two apps separately to check if the *sources* and *sinks* from both apps match. In our tests, neither FlowDroid or Amandroid were able to identify apps receiving sensitive information through intents or shared preferences. As both tools execute single app analysis, they have no information about the kind of information that can be received through those channels, and therefore not consider those as possible transmitters of sensitive information. This means that these tools are appropriate to identify apps leaking sensitive information, but not so good on identifying apps making use of that information.

In 2015, Li et al. proposed APKCombiner, a method to avoid this issue [25]. APKCombiner combines two apps in such a way that IAC channels between the apps are transformed into ICC channels (as the components that communicate are now within the same app). In this way, tools like FlowDroid or Amandroid will be able to directly trace the sensitive information flowing through the two components (that are now together in the same app). It must be noted that the app resulting from the execution of APKCombiner may not correctly execute on a system. However, the resulting app is still valid for the purpose of static analysis and transforming inter-app communication channels into inter-component communication channels within the same combined app [25].

We have used our dataset of 240 app sets to validate two different methods for collusion risk assessment. In both cases, APKCombiner joins all app components (activities, services, etc.) into a single APK, solving naming conflicts and merging both app manifest files. The generated apk file inherits the permissions from both of the apps being combined and inter-app communications between the two apps become inter-component communications. After applying APK combiner to each of the colluding app sets, we use DroidRisk [31] and Amandroid [32] (Fig. 3).

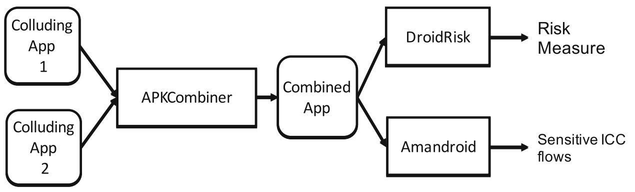

Fig. 3 Process followed for risk analysis of colluding applications. Apps are combined with APKCombiner before its analysis with DroidRisk and Amandroid

4.3.1 DroidRisk

DroidRisk measures the risk posed by an app based on the permissions it requests. Each newly requested permission (p_i) increments the risk posed by an app depending on its impact (I) and its likelihood (L). The overall risk level of an app is calculated as the sum of the risk levels of the permissions it requests (Eq. 1).

$$R_{app} = \sum_i R(p_i) = L(p_i) \times I(p_i) \tag{1}$$

The likelihood and impact of each permission were measured by analyzing two datasets of 27,274 benign apps from Google Play and 1,260 Android malware from the Malware Genome Project. The likelihood of a permission was defined as the probability of an app A being malware if that app is requesting that permission (Equation 2).

$$P(A \in malware|p_i) = \frac{P(p_i|A \in malware) \times P(A \in malware)}{P(p_i)} \tag{2}$$

The impact of requesting a permission was measured by the categories assigned to them by Android: normal and dangerous (other categories can not be requested by third party apps).

Colluding apps generally distribute the permissions they require to execute a threat. In our experiments we measured the risks created by single and combined apps.[4] Combining the permissions used by apps that communicate to measure the risk of collusion was also proposed in [7]. Figure 4 shows two boxplots representing the risk associated with single and combined apks respectively. As expected, the risk levels of combined apks is higher than the obtained by single apps. The difference in the maxima of each category can be attributed to all the combined apks requesting the `INTERNET` permission.

[4] Risk values were obtained measuring the permissions being used in the combined APK files, and not the ones requested. This is due to a bug in the latest available version of APKCombiner.

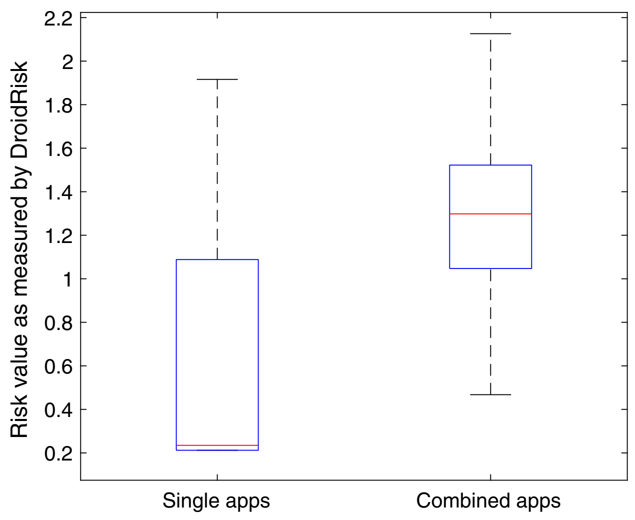

Fig. 4 Risk values obtained by DroidRisk when analysing single and combined apps

4.3.2 Amandroid

Amandroid is a static analysis tool for Android apps. It analyzes the usage of sensitive APIs in a flow and context-sensitive way across Android app components. In a nutshell, Amandroid analyzes the different information flows inside an app, checking if any sensitive API can be accessed through inter-component communication (ICC) calls. In addition, Amandroid conducts basic string analysis for inferring ICC call parameters such as the intent action. These allow Amandroid to precisely identify the ICC channels that can be used by other app components to access sensitive APIs.

In our experiments, we combine Amandroid with APK-Combiner. Amandroid is focused on ICC communications. When analyzing a single colluding app, Amandroid will find all the ICC sensitive information flows between the app, including the one that is sending information outside the app. However, it will not match that available ICC channel with the other colluding app, as it is only analysing a single app. After the APKs are combined with APKCombiner, all IAC between components of the apps would become ICC (as the components are now within the same app). Analysing the combined app, Amandroid should be able to detect now the IAC as ICC within the combined APK file. If the components identified by Amandroid in this case belong to the original different apps (APKCombiner keeps the component packages and names), we can conclude that there was an IAC between the apps.

Amandroid includes rules to detect information leakage, information injection and API misuse attacks. As our colluding apps are restricted to information theft, we have restricted our analysis to information leakage. These information leakage flows could be flagged as other confused deputy attacks.

Table 2 Results of executing Amandroid

Category	Detected	Real
Apps accessing sensitive data	199	240
Apps with ICC (Intents)	120	120
Apps with ICC (Sh. Prefs)	0	120
Apps that leak information	56	240

However, in the context of our work and due to the nature of the generated apps, we consider them as evidence of collusion (app pairs have been generated to collude to steal sensitive information).

Results obtained after running Amandroid on the 240 combined apps are shown in Table 2. Amandroid detects only 199 apps accessing sensitive information. After manually reviewing the 41 apps not flagged, we have seen that Amandroid does not consider sensitive information elements such as the user bookmarks, its navigation history, or recording from the microphone. Amandroid correctly detects the 120 that use intents to share information across components. However, it does not detect the usage of Shared Preference files as a communication method. This is an expected result, as Amandroid does not track the usage of this kind of files for communications across components. Overall, only 56 apps were detected as leaking information. The 120 apps that are using the shared preferences and the 64 apps accessing sensitive information not considered by Amandroid as such are not detected. Our results show that Amandroid could be improved in two ways: (i) by considering other possible communication channels (was already known before our experimental evaluation) and (ii) by adding the navigation history, the bookmarks and the microphone as sources of sensitive information.

4.4 Discussion

ACE allows researchers to quickly generate colluding app sets. In this way, they can focus on developing new detection methods, rather than spending their time implementing proof-of-concept colluding apps that have already been developed by other researchers.

However, as with other artificially crafted corpus, automatic generated apps should be used with care to avoid certain risks. For instance, let's imagine such corpus is the only data used to train a machine learning algorithm. In this scenario, there is a non-negligible risk of generating a model that, instead of distinguishing colluding from non-colluding apps, is distinguishing ACE generated from non-ACE generated apps. This could be avoided in three ways: (i) by avoiding features too similar within the set of generated apps (size, number of activities, etc.); (ii) by increasing the variability of code snippets and templates used in ACE; and (iii) by including the few colluding applications that have already been found in the wild [2,6,9].

5 Conclusions

In this paper we have presented ACE (Application Collusion Engine), a system capable of easily generating colluding app datasets. The datasets generated by ACE can be used by researchers to validate and compare different collusion detection proposals.

We have tested ACE by generating 480 colluding apps (240 colluding app pairs) that execute different information theft attacks. All the generated colluding app pairs were executed to verify the collusion attack was possible. Although for experimentation we have generated colluding sets consisting of two apps, ACE is capable of generating app sets of an arbitrary number of apps. We have used two different approaches to measure the risk of the generated apps. First, we have compared the risk levels obtained by single and combined applications using Droidrisk. Results show that approaches that focus on single app analysis can underestimate the risk that an app poses to a system. Approaches that analyze sensitive information flows, like Amandroid, are better at detecting collusion attacks. However, as in most collusion detection research, these tools focus only on apps using the standard ICC communications provided by Android (Intents). Adversaries may take advantage of this fact by implementing their attacks through other well known, but not yet detectable, channels such as shared preferences (up to Android 4.3), external storage o covert channels.

Researchers working on new collusion detection methods (static and dynamic analysis and operating system extensions) can easily extend ACE to quickly generate fully working apps for testing. This will enable, not only a quick validation, but also fair comparison between different proposals. As an example, [12] recently proposed an energy consumption based method to detect the usage of several covert channels for app collusion. Authors provide the source code required to implement such covert channels. Their validation experiments were executed by transmitting simple messages after a random wait for each of the channels. By adapting the source code of those channels to ACE, validations could be executed on thousands of different colluding apps that transmit real colluding messages instead.

ACE is available by request (with its source code) from the authors. Due to the risk of misuse, it has not been uploaded to public repositories.

Acknowledgements This work was supported by the UK Engineering and Physical Sciences Research Council (EPSRC) through grant EP/L022699/1.

References

1. Virusshare (2013). http://www.virusshare.com. Accessed Jan 2017
2. McAfee Labs Threats ReportJune 2016. In: Technical Report, McAfee (Intel Security) (2016). http://www.mcafee.com/us/resources/reports/rp-quarterly-threats-may-2016.pdf
3. Al-Haiqi, A., Ismail, M., Nordin, R.: A new sensors-based covert channel on android. Sci. World J. **2014**, 1–12 (2014)
4. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K.: Drebin: effective and explainable detection of android malware in your pocket. In: NDSS (2014)
5. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K.: DREBIN: effective and explainable detection of android malware in your pocket. In: 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014. The Internet Society (2014). http://www.internetsociety.org/doc/drebin-effective-and-explainable-detection-android-malware-your-pocket
6. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. ACM SIGPLAN Not. **49**(6), 259–269 (2014)
7. Asavoae, I.M., Blasco, J., Chen, T.M., Kalutarage, H.K., Muttik, I., Nguyen, H.N., Roggenbach, M., Shaikh, S.A.: Towards automated android app collusion detection. Innov. Mob. Priv. Secur. **1575**, 29–37 (2016)
8. Bhandari, S., Laxmi, V., Zemmari, A., Gaur, M.S.: Intersection automata based model for android application collusion. In: 2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA), pp. 901–908. IEEE (2016)
9. Blasco, J., Muttik, I., Roggenbach, M., Chen, T.M.: Wild android collusions. In: VirusBulletin 2016. Virus Bulletin (2016)
10. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R.: Xmandroid: a new android evolution to mitigate privilege escalation attacks. Technische Universität Darmstadt, Technical Report TR-2011-04 (2011)
11. Canfora, G., Lorenzo, A.D., Medvet, E., Mercaldo, F., Visaggio, C.A.: Effectiveness of opcode ngrams for detection of multi family android malware. In: 10th International Conference on Availability, Reliability and Security, ARES 2015, Toulouse, France, August 24–27, 2015, pp. 333–340 (2015). doi:10.1109/ARES.2015.57
12. Caviglione, L., Gaggero, M., Lalande, J.F., Mazurczyk, W., Urbański, M.: Seeing the unseen: revealing mobile malware hidden communications via energy consumption and artificial intelligence. IEEE Trans. Inf. Forensics Secur. **11**(4), 799–810 (2016)
13. Chandra, S., Lin, Z., Kundu, A., Khan, L.: Towards a systematic study of the covert channel attacks in smartphones. In: International Conference on Security and Privacy in Communication Networks, pp. 427–435. Springer (2014)
14. Dai, G., Ge, J., Cai, M., Xu, D., Li, W.: Svm-based malware detection for android applications. In: Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, New York, NY, USA, June 22–26, 2015, pp. 33:1–33:2 (2015). doi:10.1145/2766498.2774991
15. Davi, L., Dmitrienko, A., Sadeghi, A.R., Winandy, M.: Privilege escalation attacks on android. In: Information Security, pp. 346–360. Springer (2011)

16. Dimitriadis, A., Efraimidis, P.S., Katos, V.: Malevolent app pairs: an android permission overpassing scheme. In: Proceedings of the ACM International Conference on Computing Frontiers, pp. 431–436. ACM (2016)
17. Elish, K.O., Shu, X., Yao, D.D., Ryder, B.G., Jiang, X.: Profiling user-trigger dependence for android malware detection. Comput. Secur. **49**, 255–273 (2015)
18. Enck, W., Ongtang, M., McDaniel, P.: Mitigating android software misuse before it happens. The Pennsylvania State University, Technical Report, NAS-TR-0094-2008 (2008)
19. Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission re-delegation: attacks and defenses. In: USENIX Security Symposium (2011)
20. Fritz, C., Arzt, S., Rasthofer, S., Bodden, E., Bartel, A., Klein, J., le Traon, Y., Octeau, D., McDaniel, P.: Highly precise taint analysis for android applications. In: EC SPRIDE, TU Darmstadt, Technical Report (2013)
21. Han, H., Chen, Z., Yan, Q., Peng, L., Zhang, L.: A real-time android malware detection system based on network traffic analysis. In: Algorithms and Architectures for Parallel Processing—15th International Conference, ICA3PP 2015, Zhangjiajie, China, November 18–20, 2015. Proceedings, Part III, pp. 504–516 (2015). doi:10.1007/978-3-319-27137-8_37
22. Hansen, M., Hill, R., Wimberly, S.: Detecting covert communication on android. In: Local Computer Networks (LCN), 2012 IEEE 37th Conference on, pp. 300–303. IEEE (2012)
23. Hardy, N.: The confused deputy:(or why capabilities might have been invented). ACM SIGOPS Oper. Syst. Rev. **22**(4), 36–38 (1988)
24. Kim, K., Choi, M.: Android malware detection using multivariate time-series technique. In: 17th Asia-Pacific Network Operations and Management Symposium, APNOMS 2015, Busan, South Korea, August 19–21, 2015, pp. 198–202 (2015). doi:10.1109/APNOMS.2015.7275426
25. Li, L., Bartel, A., Bissyandé, T.F., Klein, J., Le Traon, Y.: Apkcombiner: Combining multiple android apps to support inter-app analysis. In: ICT Systems Security and Privacy Protection, pp. 513–527. Springer (2015)
26. Lineberry, A., Richardson, D.L., Wyatt, T.: These arent the permissions youre looking for. DefCon **18**, 2010 (2010)
27. Marforio, C., Francillon, A., Capkun, S., Capkun, S., Capkun, S.: Application collusion attack on the permission-based security model and its implications for modern smartphone systems (2011)
28. Marforio, C., Ritzdorf, H., Francillon, A., Capkun, S.: Analysis of the communication between colluding applications on modern smartphones. In: Proceedings of the 28th Annual Computer Security Applications Conference, pp. 51–60. ACM (2012)
29. Schlegel, R., Zhang, K., Zhou, X.y., Intwala, M., Kapadia, A., Wang, X.: Soundcomber: a stealthy and context-aware sound trojan for smartphones. In: NDSS, vol. 11, pp. 17–33 (2011)
30. Suarez-Tangil, G., Tapiador, J.E., Peris-Lopez, P., Ribagorda, A.: Evolution, detection and analysis of malware for smart devices. IEEE Commun. Surv. Tutor. **16**(2), 961–987 (2014). doi:10.1109/SURV.2013.101613.00077
31. Wang, Y., Zheng, J., Sun, C., Mukkamala, S.: Quantitative security risk assessment of android permissions and applications. In: Data and Applications Security and Privacy XXVII, pp. 226–241. Springer (2013)
32. Wei, F., Roy, S., Ou, X., et al.: Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 1329–1341. ACM (2014)
33. Wu, L., Du, X., Zhang, H.: An effective access control scheme for preventing permission leak in android. In: International Conference on Computing, Networking and Communications (ICNC), pp. 57–61. IEEE (2015)
34. Yue, M., Robinson, W.H., Watkins, L., Corbett, C.: Constructing timing-based covert channels in mobile networks by adjusting cpu frequency. In: Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy, p. 2. ACM (2014)
35. Zhou, Y., Jiang, X.: Android malware genome project (2012). http://www.malgenomeproject.org