FlowCog: Context-aware Semantics Extraction and Analysis of Information Flow Leaks in Android Apps

Xiang Pan, *Google Inc./Northwestern University;* Yinzhi Cao, *The Johns Hopkins University/ Lehigh University;* Xuechao Du and Boyuan He, *Zhejiang University;* Gan Fang, *Palo Alto Networks;* Yan Chen, *Zhejiang University/Northwestern University*

https://www.usenix.org/conference/usenixsecurity18/presentation/pan

This paper is included in the Proceedings of the 27th USENIX Security Symposium.

August 15–17, 2018 • Baltimore, MD, USA

ISBN 978-1-931971-46-1

FlowCog: Context-aware Semantics Extraction and Analysis of Information Flow Leaks in Android Apps

Xiang Pan
Google Inc./Northwestern University

Yinzhi Cao
The Johns Hopkins University/Lehigh University

Xuechao Du
Zhejiang University

Boyuan He
Zhejiang University

Gan Fang
Palo Alto Networks

Yan Chen
Zhejiang University/Northwestern University

Abstract

Android apps having access to private information may be legitimate, depending on whether the app provides users enough semantics to justify the access. Existing works analyzing app semantics are coarse-grained, staying on the app-level. That is, they can only identify whether an app, as a whole, should request a certain permission, but cannot answer whether a specific app behavior under certain runtime context, such as an information flow, is correctly justified.

To address this issue, we propose *FlowCog*, an automated, flow-level system to extract flow-specific semantics and correlate such semantics with given information flows. Particularly, *FlowCog* statically finds all the Android views that are related to the given flow via control or data dependencies, and then extracts semantics, such as texts and images, from these views and associated layouts. Next, *FlowCog* adopts a natural language processing (NLP) approach to infer whether the extracted semantics are correlated with the given flow.

FlowCog is open-source and available at `https://github.com/SocietyMaster/FlowCog`. Our evaluation shows that *FlowCog* can achieve a precision of 90.1% and a recall of 93.1%.

1 Introduction

Android apps, due to the nature of their functionalities, often have access to users' private information. For example, a weather app may request a user's location to provide customized weather services; a call app may obtain or import a user's phone book to ease the dialing. While these examples provide legitimate usages of private information, some apps may also misuse such information, such as stealing users' call history without their knowledge.

That said, an app needs to justify an access to users' private information with sufficient semantics available to users. For example, a weather app will clearly state that it provides local weather condition so that a user will understand its access to location information. In fact, existing researches have already started to study the semantics of an app's behaviors. For example, many past researches, such as CHABADA [19], Whyper [27] and AutoCog [28], tried to correlate an app's description, such as these in Google Play, with the permissions that the app asks for.

However, existing approaches [19, 27, 28, 37] are coarse-grained, staying on the app level. They can identify whether an app should have access to a certain piece of private information, but cannot justify whether the access should happen under certain context. For example, an app may have two data flows[1] [12,15,18,22,24,33] accessing private information, one providing a customized service with user's knowledge, e.g., a pop-up window, but the other hiding secretly in background and sending information to the Internet without user's knowledge. Apparently, the former is legitimate with sufficient semantics, which we call *positive* in the paper, but the later is not, hence defined as *negative*.

In this paper, we propose an automated, flow-level system, called *FlowCog*, to extract and analyze semantics for each information flow of an Android app. *FlowCog* is fine-grained, because it extracts flow-specific semantics called *context*, e.g., information in a registration interface and a pop-up window, and correlates the context with the information flow. While intuitively simple, the challenge of *FlowCog* lies in how to extract such context, i.e., *FlowCog* needs to establish a relationship between semantics embedded deeply in an app with each information flow.

The key insight of *FlowCog* is that flow contexts are embedded in these Android GUIs, such as views, which have direct control over the flow. For example, if the information flow is that an Android app sends a phone number to the Internet after the user clicks a submit button, such as the running example shown in Section 2, the

[1]We use the following two terminologies, "information flow" and "data flow", interchangeably in this paper.

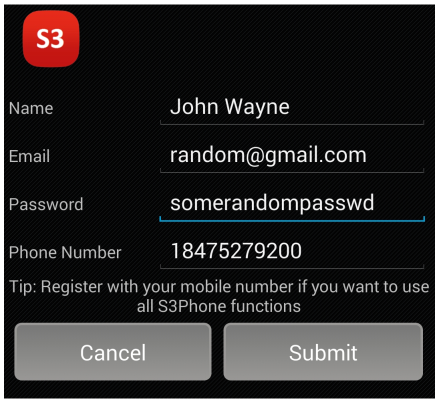


Figure 1: Registration Interface of S3 World Phone App

flow context will be in the view that has the submit button. Particularly, *FlowCog* performs static analysis that connects UI views, such as button and checkbox, of Android apps with given information flows via control and data dependencies. Then, *FlowCog* extracts flow contexts, e.g., texts and images, embedded in UI views via a mostly static approach with an optional dynamic component.

Once flow contexts are extracted, *FlowCog* distills texts from images via image recognition, and then analyzes texts including these extracted from images using an NLP module. Lastly, *FlowCog* adopts two classifiers to determine the correlation between flow contexts and the flow. A high correlation indicates that the flow is positive, i.e., the Android app provides sufficient semantics for the flow, and a low correlation means not.

We have implemented a prototype of *FlowCog*. Our evaluation on the prototype against 2,342 flows extracted by FlowDroid [12] shows that *FlowCog* has an overall precision of 90.1% and a recall of 93.1%. We also show that flow contexts can provide more justifications, i.e., 10% more in terms of accuracy, than app-level semantics alone. *FlowCog* is open source and available at the following repository: `https://github.com/SocietyMaster/FlowCog`.

2 Overview

In this section, we give an overview of *FlowCog* via a running example, called S3 World Phone app (called S3 app for short), which allows users to make phone calls world-wide. The S3 app sends a user-provided phone number to its own server after the user sees a registration page shown in Figure 1 and presses the "Submit" button. This flow, from the phone number to the Internet, is positive, because the app provides sufficient semantics, such as keywords "Phone Number" and "mobile number", so that the app user can acknowledge and authorize the flow.

What *FlowCog* does is to extract contexts for each information flow found by existing static or dynamic analysis and classify the flow as either positive or negative based on the extracted contexts. Specifically, such process, shown in Figure 2, can be broken down into four steps: (*i*) finding information flows of an Android App, (*ii*) finding special statements called *activation event* and *guarding condition* via control dependency and associated views (called view dependency) for each information flow, (*iii*) finding and extracting contexts, e.g., texts and images, from the aforementioned two special statements via data dependency, and (*iv*) determining the correlation between the flow and the contexts via Natural Language Processing (NLP) technique. Note that in the third step, *FlowCog* can optionally rely on a dynamic analysis that instruments the target app, performs UI exercise, and outputs key values of certain variables.

Now we use our running example to illustrate how the four-step process works. First, *FlowCog* will rely on existing approaches, such as FlowDroid, to find information flows of the Android app. The phone number leak of the S3 app, shown in Figure 3, starts from $TelephonyManager.getLine1Number()$, i.e., the source, in Block 1, and flows to $HttpClient.execute()$, i.e, the sink, in Block 4. Details are as follows. The phone number is first stored in an *EditText et_regist_phone* (Block 1), read by the *getText* method (Block 2), and then loaded by $S3ServerApi.performRegistration$ as a parameter (Block 3). Then, the $S3ServerApi.postData$ method reads the phone number and sends it to the Internet via $HttpClient.execute$, i.e., the sink (Block 4). All statements are marked in Figure 3 via circled numbers in sequence following the information flow.

Second, *FlowCog* finds two special statements, called *activation event* and *guarding condition*, which are related to the information flow via control and view dependency and can be used to extract flow contexts. The S3 app contains examples of both special statements. Block 5 shows an example of *activation event*, because the $performRegistration$ method in Block 6 is activated by an onClick event. The second statement in Block 2 shows an example of *guarding condition*, because this statement prevents the phone number leak if the condition is unsatisfied. In this example, the statement only allows the phone number leak if the inputted password is strong enough to pass the complexity test.

Particularly, here is how *FlowCog* finds both activation events and guarding conditions for the S3 app. *FlowCog* finds that the $performRegistration$ method in Block 6, an activation event, is connected with Block 2, a block in the target data flow, via control dependency. Then, *FlowCog* finds additional special statement, e.g., another activation event in Block 5, based on corresponding views, e.g., Button *bt_regist_submit*, associated with the found activation event, e.g., $performRegistration$—such process is defined as view dependency in this paper. Following both control and view dependency, *FlowCog*

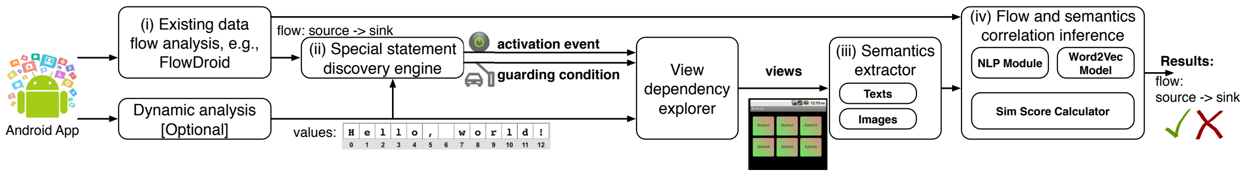
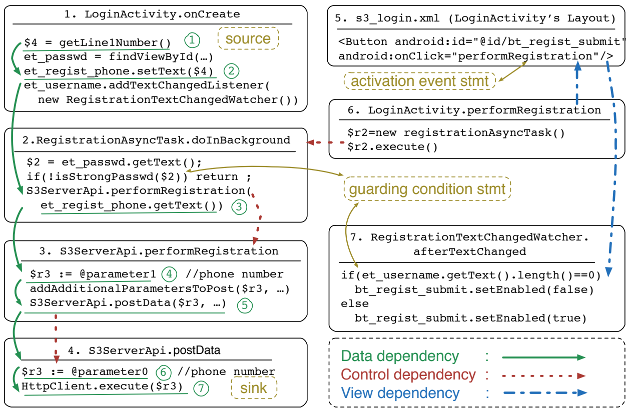


Figure 2: *FlowCog* Architecture



```
1. LoginActivity.onCreate
$4 = getLine1Number() ①        source
et_passwd = findViewById(…)
et_regist_phone.setText($4) ②
et_username.addTextChangedListener(
  new RegistrationTextChangedWatcher())

2.RegistrationAsyncTask.doInBackground
$2 = et_passwd.getText();
if(!isStrongPasswd($2)) return ;
S3ServerApi.performRegistration(
  et_regist_phone.getText()) ③

3. S3ServerApi.performRegistration
$r3 := @parameter1 ④ //phone number
addAdditionalParametersToPost($r3, …)
S3ServerApi.postData($r3, …) ⑤

4. S3ServerApi.postData
$r3 := @parameter0 ⑥ //phone number
HttpClient.execute($r3) ⑦        sink

5. s3_login.xml (LoginActivity's Layout)
<Button android:id="@id/bt_regist_submit"
android:onClick="performRegistration"/>

6. LoginActivity.performRegistration
$r2=new registrationAsyncTask()
$r2.execute()

7. RegistrationTextChangedWatcher.
        afterTextChanged
if(et_username.getText().length()==0)
  bt_regist_submit.setEnabled(false)
else
  bt_regist_submit.setEnabled(true)
```

Figure 3: Simplified Code Blocks of S3 World Phone App

can also find guarding conditions, such as the *if* statement in Block 2 and Block 7.

Third, *FlowCog* finds and extracts contexts, e.g., texts and images, starting from activation events and guarding conditions via data and view dependency. From Block 5, i.e., the activation event, *FlowCog* directly finds the Submit Button and the surrounding texts, i.e., "Submit", via view dependency. From the second statement in Block 2, *FlowCog* performs a data flow analysis upon \$2 and finds *et_passwd*, a text field, the surrounding texts, "Password". From the guarding condition in Block 7, *FlowCog* finds the user name field. In all scenarios, *FlowCog* will find surrounding texts, such as "Tips: Register with mobile number ...".

Lastly, *FlowCog* determines the correlation between the found flow contexts and the target flow. Specifically, *FlowCog* processes the texts, e.g., "Submit" and "Tips: Register with mobile number ...", removes stop words, and converts the words into a vector using an NLP module. At the same time, *FlowCog* processes API documents related to the sink and source, i.e., $TelephonyManager.getLine1Number()$ and $HttpClient.execute$ with the same method into a vector. Then, *FlowCog* feeds both vectors into two types of classifiers, one learning-based and the other learning-free, combines the outputs using linear regression, and calculates an overall score. In this example, the score is high, thus the flow being considered as positive, because "Tips: Register with mobile number ..." is related to the source and "Submit" related to the sink.

```
1  LoginActivity.onCreate(...)
2  registrationAsyncTask.doInBackground()
3  S3ServerApi.performRegistration(...,
       et_regist_phone.getText())
4  S3ServerApi.postData($r3, ...)
```

Figure 4: Call Path for the Data Flow in Figure 3

3 Design

In this section, we present the details of each component of *FlowCog*'s architecture in Figure 2. Information flow analysis, i.e., step (*i*) in Figure 2, is skipped, because we just use existing ones, such as FlowDroid. We first present the special statement discovery engine in Section 3.1, which finds both activation events and guarding statements, and view dependency explorer in Section 3.2. Then, we show how to extract semantics from views and other places in Section 3.3 and correlate extracted semantics, i.e., flow contexts, with flows in Section 3.4. Lastly, we introduce an optional dynamic analysis component in Section 3.5.

3.1 Special Statement Discovery Engine

Special statement discovery engine finds activation events and guarding conditions given a data flow. The reason of finding these two special statements is that they have direct control over the given data flow: Activation events decide whether to trigger the data flow and guarding conditions determines whether the source flows to a sink or other places. That is, semantics associated with these two special statements will influence users' decision and perception on the data flow. For example, the activation event in Block 5 of Figure 3 is a submit button, which directly controls the phone number leak and gives users semantics. Next let us discuss these two special statements separately.

3.1.1 Activation Event

Intuitively, an activation event, e.g, the *onCreate* and *performRegistration* methods of the *LoginActivity* class in our running example (Figure 3), is a callback method that initiates a given flow. In other words, the flow happens only after the activation event is invoked. Now, we give a formal definition of an activation event.

Definition 1. (Activation Event) *Given a data flow, we define an event callback p_e as an activation event if there exists a path $p_e ... p_k$ in the call graph of the target app where p_k is a statement in the flow's call path ($p_{src} ... p_k ... p_{sink}$). Note that a call path of a given data flow is defined as all the caller statements, in the calling sequence, of methods containing each statement in the data flow.*

Now let us discuss how *FlowCog* finds all the activation events. First, *FlowCog* extracts all the registered, possible event callback methods and stores them into a list, called *reg_call_lst*. Take UI events for example. *FlowCog* extracts callback methods from both codes and layout files. Specifically, *FlowCog* parses the app's codes to identify all those event listener registration statement (e.g., setOnClickListener(...)) and then gets the callbacks by extracting the name of argument class. Then, *FlowCog* parses the layout files and saves the values of those event attributes (e.g., onClick attribute). Similarly, *FlowCog* finds lifecycle event callbacks by looking at subclasses of corresponding lifecycle related classes, such as *Activity*, and finding overridden lifecycle callbacks, such as *onCreate*.

Then, *FlowCog* generates call paths for a given data flow, e.g., the call path in Figure 4 for the data flow in Figure 3, and performs Algorithm 1 to find its activation events. Particularly, *FlowCog* first reverses the call path for easy processing (Line 3), and then goes through every statement in the call path to see whether it is in the *reg_call_lst* (Line 4–10). If so, *FlowCog* adds the statement in the result set (Line 6); if not and the statement is the first in the method compared with others in the call path, *FlowCog* adds the parent of this statement in a queue for further processing. Note that *FlowCog* only adds the first statement because other statements will share the same parent with the first. Next, *FlowCog* goes through every added statements in the queue (Line 11–21) until the queue is empty. For each statement in the queue, *FlowCog* determines whether it is in the *reg_call_lst* (Line 13–14). If so, *FlowCog* adds the statement in the result set; if not and the statement is unvisited before (Line 15), *FlowCog* goes backward through the call graph and puts its parent in the queue (Line 16–18).

3.1.2 Guarding Condition

Intuitively, a guarding condition of a given data flow is a conditional statement, e.g., *if* statement, which may affect the execution of the data flow. For example, if one branch of an *if* statement allows the data flow but another terminates the flow, we consider such *if* statement as guarding condition—both *if* statements in Blocks 2 and 7 in Figure 3 are such examples. We now formally define guarding condition in Definition 2.

Algorithm 1 The Algorithm of Finding Flow's Activation Event Statements

```
Input:  Data Flow's Call Path: callPath
        Call Graph: callGraph
Set findActivationEvent(callPath, callGraph):
 1: rs = createNewStmtSet()
 2: queue = createNewStmtQueue()
 3: reverse(callPath)
 4: for stmt in callPath do
 5:     if isInvokeStmt(stmt) and isInReg_Call_List(stmt) then
 6:         rs.add(stmt)
 7:     else if isFirstStmt(stmt) then
 8:         queue.add(parent)
 9:     end if
10: end for
11: while !queue.isEmpty() do
12:     stmt = queue.pull()
13:     if isInvokeStmt(stmt) and isInReg_Call_List(stmt) then
14:         rs.add(stmt)
15:     else if !isVisited(stmt) then
16:         method = getMethodOfStmt(stmt)
17:         for parent in callGraph.getCallerStmtsOf(method) do
18:             queue.add(parent)
19:         end for
20:     end if
21: end while
22: return rs
```

Definition 2. (Guarding Condition) *Given a data flow $n_{source} ... n_k ... n_{sink}$, for any n_k, we define a conditional statement c_e—at least one branch of which does not contain n_k—as a guarding condition if either of the following is satisfied:*

(1) c_e and n_k are in the same basic block, or connected in the interprocedural Control Flow Graph (iCFG);

(2) c_e controls the activation events of the data flow via view dependency, i.e., c_e and the activation event are in the same view.

Based on the definition, there are naturally two phases to find all guarding condition statements. In the first phase, *FlowCog* identifies guarding conditions that are directly connected with the data flow in the iCFG; and then, in the second phase, *FlowCog* identifies guarding conditions that are connected with the data flow's activation events.

Algorithm 2 shows the first phase in which *FlowCog* iterates all the statements in the data flow reversely. During each iteration, *FlowCog* extracts two consecutive statements, *prevStmt* and *curStmt*. If these two statements are in the same method, *FlowCog* searches the guarding condition statements, *stmt*, from those statements, such that there exists a path $P = prevStmt_{...}stmt_{...}curStmt$ in the method's control flow graph (Line 9–10). If these two statements are from different methods, but *prevStmt* is the caller of *curStmt*'s method, *FlowCog* search the guarding condition statements from those statements in *curStmt*'s method that can reach *stmt* (Line 11–12). If none of the following are satisfied, i.e., the method of *curStmt* is a callback method, *FlowCog* searches the statements that can reach *curStmt* in the program's inter-procedure control flow

Algorithm 2 The Algorithm of Finding Guarding Condition

```
Input:
    Flow Data Path: path
    Interprocedure Control Flow Graph: iCfg
Set findGuardingCondition(path, graph):
 1: $rs = createNewStmtSet()$
 2: for ($i = path.size() - 1$; $i >= 0$; $i$–) do
 3:     if $i == 0$ then
 4:         $findGCHelper(path.get(0), null, iCfg, rs)$
 5:     else
 6:         $prevStmt = path.get(i-1)$
 7:         $curStmt = path.get(i)$
 8:         $method = getMethodOfStmt(curStmt)$
 9:         if $fromSameMethod(prevStmt, curStmt)$ then
10:             $findGCHelper(curStmt, prevStmt, iCfg, rs)$
11:         else if $isInvokeStmt(prevStmt)$ and
              $method == getInvokedMethod(prevStmt)$ then
12:             $findGCHelper(curStmt, method.getFirstStmt(), iCfg, rs)$
13:         else
14:             $findGCHelper(curStmt, null, iCfg, rs)$
15:         end if
16:     end if
17: end for
18: return $rs$
```

graph (Line 13–14).

We then discuss the search algorithm mentioned in previous paragraph in Algorithm 3. Specifically, the algorithm starts from a *target* node, i.e., the *curStmt* in Algorithm 2, and conducts a reverse breadth-first search (Line 16–18) in the iCFG to find conditional statement. For each found condition statement, the algorithm additionally checks whether this statement has a child node that *cannot* reach the *target* node (Line 9–14). If there exists such child, the conditional statement is a guarding condition.

Next, *FlowCog* finds all the conditional statements that control activation events of the given data flow in the second phase. Specifically, *FlowCog* finds all the view objects that registered the activation events and then searches for the following control statements in the found views: (*i*) *View.setEnabled(boolean)*, (*ii*) *View.setClickable(boolean)*, (*iii*) *View.setVisibility(boolean)*, and (*iv*) *View.setLongClickable(boolean)*. *FlowCog* again performs Algorithm 2 starting from all the found control statements to identify additional guarding conditions. Consider our running example in Figure 3 again. The method *LoginActivity.performRegistration(...)* is an activation event, and *FlowCog* finds corresponding guarding conditions related to the activation event by identifying the view, i.e., Button *bt_login_submit*, and then performs Algorithm 2 upon *setEnabled* in the view's code at Block 7 of Figure 3.

3.2 View Dependency Explorer

After *FlowCog* finds two special statements for a given data flow, it finds Android views related to the data flow so as to extract semantics. We call such relationship between views and the data flow as *view dependency*.

Algorithm 3 The Algorithm of Finding Guarding Condition Helper Method

```
Input:
    Target Statement: target
    End Statement: endStmt
    Interprocedure Control Flow Graph: iCfg
    Guarding Condition Result Set: rs
void findGCHelper(target, endStmt, iCfg, rs):
 1: $queue = createNewStmtQueue()$
 2: $queue.add(target)$
 3: while $!queue.isEmpty()$ do
 4:     $stmt = queue.poll()$
 5:     if $stmt == endStmt$ then
 6:         $continue$
 7:     else if $!isVisited(stmt)$ then
 8:         if $isConditionStmt(stmt)$ then
 9:             for $child$ in $iCfg.getSuccessors(stmt)$ do
10:                 if $!canReachStmt(child, target)$ then
11:                     $rs.add(stmt)$
12:                     $break$
13:                 end if
14:             end for
15:         end if
16:         for $parent$ in $iCfg.getPredecessors(stmt)$ do
17:             $queue.add(parent)$
18:         end for
19:     end if
20: end while
```

Specifically, view dependency can be classified into the following three categories.

- Data flow related. A view can be dependent on the given data flow directly. For example, if the source of the data flow is obtained from a view (e.g., EditText), such dependency exists.
- Activation event related. If an activation event of the given data flow belongs to a view, e.g., registered as a event handler, we consider such dependency exists.
- Guarding condition related. If a view's attribute values (e.g., EditText.getText() or CheckBox.isChecked()) could change the conditional result in guarding conditions of the given data flow, we consider such dependency exists.

The view dependency problem can be formalized into another data flow analysis. The sources in this analysis are all the possible views, and the sinks are the aforementioned three scenarios, i.e., the given data flow, its activation events, and its guarding conditions. Now, let us explain in details how *FlowCog* obtains these sources and sinks.

First, *FlowCog* obtains all the sources by going through all the view definitions, either static or dynamic. *FlowCog* parses layout files that statically define views and treats all the *findViewById(...)* and *inflate(...)* invoke statements related to these views as source. In addition, *FlowCog* adopts a manually created list about all possible View classes from the Android documentation and finds all the *new* statements that create an object with these classes—these statements are treated as source as well.

Second, *FlowCog* obtains all the sinks based on the aforementioned dependency categories. Statements in the given data flow and guarding conditions are added directly to the sink list. Then, *FlowCog* searches through the entire program for all activation events' registration statements, e.g., *setOnClickListener* corresponding to *onClick*, and adds these registrations to the sink list. Note that an activation event may be defined in layout files—in such case, the data flow analysis is simplified to a direction association of the activation event and the view defined in the layout file.

3.3 Semantics Extraction

The next step of *FlowCog* is to extract semantics, e.g., flow contexts, from views that has a dependency with a given data flow. Besides depended views, we find that semantics could also exist in the app's description on Google Play and other views in the same visible layouts. We now discuss how to extract such semantics.

3.3.1 Semantics Extraction from App Description

An app's description, available in Google Play for crawling, is what a user sees even before using the app—this is also what existing approaches use to extract app semantics [19, 27, 28, 37]. Apart from descriptions in Google Play, if an app is provided without any descriptions, e.g., malicious apps collected by security researchers, *FlowCog* will treat texts from the app's string resource file as a substitute of descriptions.

3.3.2 Flow Context Extraction from Views

There are two types of flow contexts: those from views that have dependencies with a given data flow, and those from other Views in the same Layout of the Depended View. Let us discuss these two separately.

First, semantics exist in views that have dependencies with a given data flow, thus directly affecting the flow's execution. For example, in Figure 3, the Button view will control the program in deciding whether to send out the phone number, and its text, i.e., the "submit" word, is the semantics about sending behavior. For another example, an "alert" Dialog view asking for user's permission for sharing her location decides whether the location is sent to the server, and provides semantics in its text to users.

The semantics extraction for such views has two steps. (i) *FlowCog* resolves the identifiers of such views. Specifically, *FlowCog* resolves the value of *findViewById(...)* and *inflate(...)*'s argument both statically via searching the definition of the parameter backward in the iCFG and dynamically via an optional dynamic analysis in Section 3.5. Note that based on our evaluation, 97.6% of values can be resolved statically. (ii) *FlowCog* extracts semantics related to the views. Specifically, *FlowCog* finds all the invoke statements with their base object as the view, and the invoked method as one of the following *<init>(...)* (the constructor method's name in Jimple), *setTitle(...)* and *setTexts(...)*. Then, *FlowCog* resolves the parameter value of the aforementioned methods following the same way as it does for the view's identifier in previous step. Again, in most cases, i.e., 94%, such values can be resolved statically; otherwise, *FlowCog* relies on the optional dynamic analysis to resolve values.

Second, besides the depended view, semantics from other adjacent views in the same layout may also be flow contexts, because a user-visible screen may contain multiple views from the same layout. "Tip: Register with your mobile number" in Figure 1 is such an example. Such semantics extraction has three steps. (i) *FlowCog* resolves the layout that the depended view locates at. Specifically, *FlowCog* looks at the second parameter of *setContentView*() method in which the first parameter is the target depended view. (ii) *FlowCog* finds other views inside the same layout by looking at other *findViewById*() and *inflate*() calls as well as all *new* statements that create dynamic views. (iii) *FlowCog* extracts semantics from other views just as what it does for the target depended view.

3.3.3 Flow Context from View's Layout

Besides views, the layout file of the view having dependency with the given data flow may also contain other resources, such as texts and images, which could provide semantics. We divide the resource types into four categories: (*i*) texts, (*ii*) text images, (*iii*) images without any texts, e.g., email and phone icons, and (*iv*) non-image fragments, e.g., maps. Now let us discuss how to extract semantics from each category.

First, for text resource, *FlowCog* extracts the values of *android:text* and *android:hint* attributes in the layout file. If the value is not a string but an identifier of other resources (e.g., string/msg), *FlowCog* further analyzes the corresponding resource files to resolve the string value and finds the string value of such identifier.

Second, for image resource, *FlowCog* extracts *android:background* attribute in the layout file. Additionally, *FlowCog* also extracts the *android:src* attribute of all image views, e.g., *ImageButton*. All the images are first fed into Optical Character Recognition (OCR) engine to extract obvious texts.

Third, *FlowCog* also adopts Google Image to analyze the topics of images extracted in previous step. Specifically, *FlowCog* stores each image as a URL, uploads the URL to Google Image's server, and uses a headless browser to obtain a result returned by Google. Note that because Google Image restricts the number of uploaded image from each IP address for a given interval, *FlowCog* only uploads images when the OCR engine cannot extract texts from the image.

Lastly, for non-image fragments, *FlowCog* re-

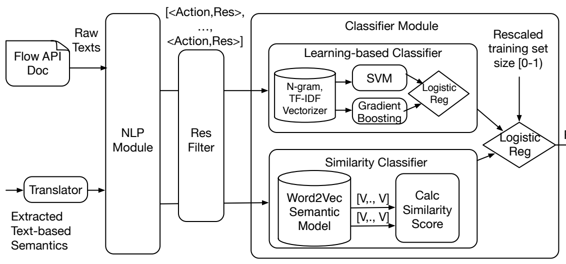


Figure 5: Flow and Semantics Inference in *FlowCog*

lies on a manual-curated list to extract semantics. Take Google Map for example. We specify two pairs of fragment name and semantics, (e.g., <com.google.android.gms.maps.SupportMapFragment, map>, <com.google.android.maps.MapView, map>) to represent a map object in the list, so that when *FlowCog* finds this fragment in a layout file or related code, a "map" semantics will be added.

3.4 Flow and Semantics Correlation Inference

In this section, we give an overview about how *FlowCog* infers the correlation between a given data flow and the extracted semantics. Because we do not claim any contributions in this component, we leave the detailed design in the Appendix. Figure 5 illustrates an overview of the design of this inference engine, which takes the flow and the text-based semantics as input and outputs whether these two are correlated. Specifically, in this part, *FlowCog* extracts the documents associated with the flow's APIs (e.g., $getLine1Number()$ and $HTTPClient.execute()$), and feeds the documents as well as previously extracted text-based semantics—with translation to English language if necessary—into an NLP module. The NLP module cleans the raw texts by converting them into a list of action-resource pairs (e.g., <"synched", "cloud">) using an NLP parser. After that, resource filter will filter those less-informative pairs generated from API doc and feed all the remaining ones into two classifiers, one learning-based and the other learning-free, and *FlowCog* will calculate a score based on the results from these two classifiers using logistic regression. We now discuss these two classifiers.

On one hand, the learning-based classifier vectorizes action-resource pairs into a numeric feature using bag-of-words [10] and TF-IDF [7]. Each element in the feature vector indicates the importance of a word or action-resource pair in identifying the target data flow. Then, two machine learning (ML) modules, namely gradient boosting and linear SVM, will take the feature vector as input for both training and prediction. Note that we choose these two ML modules because they preform the best among all the classifiers that we evaluated. The prediction results including confidence scores from these two ML modules are combined by another logistic regression module.

On the other hand, the learning-free classifier, i.e., the similarity one, measures the similarity between the action-resource pair lists from the flow's API documents and the extracted text-based semantics. Specifically, *FlowCog* converts both lists into a vector representation of words, called word embedding. (Word embedding can represent words in a continuous vector space, where semantically similar words will be mapped to nearby points.) Then, *FlowCog* transforms each action-resource pair in both lists to a vector through Word2Vec model [26], one of the most popular predictive model for learning word embedding from raw texts. Lastly, *FlowCog* calculates a similarity score between two lists from the flow's API documents and the extracted text-based semantics to represent the correlation between the given data flow and the extracted texts.

3.5 Optional Dynamic Analysis

FlowCog supports an optional dynamic analysis module to perform a dynamic value analysis and output certain strings and view IDs that cannot be resolved statically. Based on our observation, only 5.3% statements belong to such category. The dynamic analysis works in three steps.

First, the dynamic analysis instruments Android app by identifying all the text-setting statements and printing the values their parameters as well as the target text-setting statement's location immediately before each text-setting statement. The text-setting statements that we currently instrumented are listed as follows: *setTitle(...)*, *setText(...)*, *setMessage(...)*, *setPositiveButton(...)*, *setNegativeButton(...)* and *setButton(...)*.

Second, we adopt a customized version of AppsPlayground [29] to install the app on emulator and automatically explore the app dynamically. In particular, our customized AppsPlayground adopts an image processing approach to identify clickable elements and sends event signals to increase the exploring coverage. Each app is set to be explored for at most 20 mins.

Lastly, during the dynamic app exploration, when any text-setting statement is encountered, its string argument value as well as the statement's location will be printed out. After execution, these logs will be extracted and stored in a NoSql database. The key for each record is the app's name and the statement's location, while the value include the texts associated with the corresponding statements' arguments. During static analysis, if *FlowCog* encounters a string argument whose value cannot be resolved, it will lookup the database built in dynamic analysis.

Table 1: Lines of Code (LoC) of Different Components of *FlowCog*

Component	Language	LoC
Flow-related Semantics Extraction	Java	∼12,000
Classifiers	Python	∼3,000
Dynamic Analysis	Python, Java	∼1,000
Misc	Python	∼500
Total	Java, Python	∼16,500

4 Implementation

Now we discuss the implementation of *FlowCog* in this section. *FlowCog* involves ∼16,500 Lines of Code (LoC) in total, excluding any third-party libraries, such as FlowDroid, Soot, and Stanford parser. A detailed breakdown of each component can be found in Table 1. The semantics extraction part, such as finding views, activation events and guarding conditions, contains ∼12,000 LoC, the part about correlating semantics and flows, i.e., multiple classifiers, contains ∼3,000 LoC, our dynamic analysis ∼1,000 and others ∼500. We then discuss details of each component.

First, as discussed, we adopt FlowDroid, a precise and efficient Java-implemented static analysis system, to discover all information flows. All analysis steps operate on Jimple intermediate representation (IR) [32], a typed 3-address IR suitable for optimization and easy to understand. *FlowCog* uses Soot framework [23] to transform an app into Jimple codes, a widely used Java optimization framework. In text extraction engine, *FlowCog* also needs to run data flow analysis to find flow's related views. Such data flow analysis component is also based on the taint analysis framework provided by FlowDroid.

Second, we implement a crawler using BeautifulSoup [1] to crawl API documents for methods associated with each flow. Then we use Stanford Parser Wrapper [4], a Python wrapper of Stanford Parser, to cleanse these raw texts, transform them into a set of valid none-verb pairs, serving as the inputs for classifiers. Before feeding texts into classifier, we use mtranslate package [2], a Python wrapper of Google Translate API, to translate non-English texts into English. For learning-based classifier, *FlowCog* uses Python's Scikit-learn library [6], which integrates all the machine learning modules we have used in our implementation and evaluation. As for the similarity classifier, *FlowCog* chooses Word2Vec, a popular computationally-efficient predictive model for learning word embeddings.

Lastly, we use apktool [8] to decompile Android apk files. Then we write Python scripts to parse the XML resource files extracted from decompiled apk files. To extract texts from image, we adopt pytesseract package [5], a Python wrapper for google's Tesseract-OCR, one of the most popular open-source OCR tools. For dynamic analysis, we write a Soot-based Java program in ∼400 lines of codes to automatically instrument apps and then manage and customize AppsPlayGround [29] with ∼600 lines of Python codes to dynamically explore the instrumented apps.

5 Evaluation

In this section, we evaluate *FlowCog* by answering the following four research questions.

- RQ1: How accurate is *FlowCog* in identifying positive and negative flows, i.e., correlating Android app's semantics and each flow?
- RQ2: How much does flow contexts, e.g., semantics in apps' GUI, improve the overall accuracy of *FlowCog*?
- RQ3: How does *FlowCog*'s classification algorithm compare with other alternative, naïve approaches?
- RQ4: How effective is *FlowCog* in extracting flow contexts?

5.1 Experiment Setup, Dataset and Ground Truth

We run all the experiments on a Ubuntu 14.04 server with Intel Xeon 2.8G, 16 cores CPU and 32G memory. The overall dataset contains 6,000 benign and malicious apps. All the 4,500 benign apps are randomly crawled from Google Play and 1,500 malicious ones are randomly selected from Drebin dataset [11, 25]. FlowDroid with its default setting, i.e., flow- and context-sensitive, is used as the existing static analysis tool to extract information flows—we run FlowDroid on each app for 20 mins and then terminate it if no results are outputted. In the end, 1,299 benign apps terminate successfully, and 361 of them generate 1,043 flows; 586 malicious apps terminate successfully, and 255 of them generate 1,299 flows. The sizes of apps range from 16.9KB to 51.9MB.

Note that we realize that some limitations of FlowDroid, such as low termination ratio and lack of inter-component analysis, may have impacts on the final results. We did try to run FlowDroid for a longer time, such as four hours on a small set of unfinished apps—it turns out that FlowDroid cannot finish analyzing these apps either. We would like to emphasize that because the flows found by FlowDroid contain all possible pairs of sources and sinks, we believe that we have already tested *FlowCog* on varieties of flows. In addition, *FlowCog* can be combined with any other static or dynamic analysis tools outputting information flows. Because FlowDroid is the most popular and open-source static analysis tool, we rely on FlowDroid in our evaluation.

Next, we present how to obtain the ground truth for the dataset. We ask three graduate students to manually annotate each flow of Android apps as either *positive*, i.e., the app provides enough semantics for the flow, or *negative*, i.e., the app does not provide enough semantics. The details of the manual annotation work as fol-

Table 2: Manually-annotated Ground Truth and Overall Performance of *FlowCog* against the Ground Truth

App Type	# of Apps	# of Apps with Flows	# of Total Flows	# of Positive Flows	# of Negative Flows	TP	TN	FP	FN	Precision	Recall	Accuracy
Benign App	1,299	361	1,043	688	355	352	197	38	18	90.3%	95.1%	90.7%
Malicious App	586	255	1,299	675	624	312	259	35	31	89.9%	91.0%	89.6%
Overall	1,885	616	2,342	1,363	979	664	456	73	49	90.1%	93.1%	90.2%

lows. Each student is provided with information flows, Android apps, and app descriptions. We instruct the student to install Android apps, look at app descriptions and each information flow in the context of the app, and then infer whether the information flow as positive or negative based on their own knowledge.

The final ground truth results are determined by a majority vote of three students. In practice, all the 2,342 flows are unanimously annotated by the three students, which indicates that people have very few discrepancies in understanding semantics. In total, they have spent around 150 hours to annotate all these flows. Now let us describe the ground truth results in Table 2. Among the 1,043 flows from benign apps, 688 of them are positive and 355 are negative. As for the 1,299 flows from malicious apps, 675 are annotated as positive and 624 are negative. We randomly select half of the apps and use flows from these apps (650 positive flows and 450 negative flows) as training set and the remaining 1,242 flows as testing set.

5.2 RQ1: Precision, Recall and Accuracy

In this research question, we measure *FlowCog*'s *true positives* (TP), *true negatives* (TN), *false positives* (FP) and *false negatives* (FN) based on our manually annotated ground truth. From TP, TN, FP and FN, we further calculate the *precision*, *recall* and *accuracy*. Precision is defined as $TP/(TP+FP)$, recall as $(TP/(TP+FN)$, and accuracy as $(TP+TN)/(TP+TN+FP+FN)$.

Table 2 illustrates the evaluation results of *FlowCog* in accuracy. The overall precision, recall and accuracy are 90.1%, 93.1% and 90.2% respectively. *FlowCog*'s accuracy, i.e., 90.7%, on benign apps is slightly higher than one on malicious apps, i.e., 89.6%. The major reason is that malicious apps have higher false negative. Our manual analysis shows that many of those are caused by inefficient training set.

We further break down the overall accuracy of *FlowCog* based on the used permissions, e.g., Location and SMS, and then calculate each permission category's accuracy. Specifically, each flow is categorized based on its source and sink's permissions respectively. Take a flow flow "getLongitude(...) -> sendTextMessage(...)" as example. This flow is counted in both Location and SMS permission categories. Note that many permissions, e.g., Audio and Camera, are not present in our evaluation dataset.

Table 3 shows the detailed break-down results of accuracy based on permissions. Top six rows show source permissions, and bottom two rows sink permissions. There are two things worth noticing here. First, the general trend excluding some exceptions noted below is that the larger training data *FlowCog* has, the better accuracy results we can get for *FlowCog*. In the source permission categories, "Credential" has the highest accuracy while "Calendar" the lowest. In the sink permission categories, the accuracy number in "Internet" category is higher than the one in "SMS". Second, flows that have different semantics presentations have a lower accuracy than these that do not. Take flows with a "Location" permission for example. Such flows can be interpreted in many different ways, such as "map", "location", and "local weather". Hence the accuracy for "Location" is lower than that for "Phone Number", which is usually represented in literal.

5.3 RQ2: Effectiveness of Flow Contexts

In this subsection, we show that flow contexts can improve *FlowCog*'s accuracy in classifying positive and negative flows. Particularly, we compare *FlowCog* with approaches that takes (*i*) only apps' descriptions, (*ii*) only flow contexts, (*iii*) apps' description and all the flow's contexts, and (*iv*) apps' description and the context for only the target flow (i.e., *FlowCog*). The purpose is to show that contexts for the target flow can provide more information in correlating the flow with app's semantics, but other flow's contexts will have a negative impact.

The right four columns in Table 4 show our evaluation results. The accuracy for *FlowCog* is the highest among all other possibilities. The results show that flow contexts provide more information than the app descriptions, and at the same time app descriptions provide a background for flow contexts—therefore, the combination of these two provides a good result for *FlowCog*. At the same time, the results also show that other flows' contexts may provide negative impacts on the overall accuracy. Specifically, the false positive is very high when we include other flows' contexts, because such contexts may be unrelated to the target flow.

5.4 RQ3: Comparison with Alternative Classification Approaches

In this subsection, we would like to justify why we make such a choice in designing *FlowCog*. Specifically, we want to compare the followings: (i) learning-based model vs. learning-free model vs. the hybrid model

Table 3: Flow Classification Accuracy by Permissions

Permission	Number	TP	TN	FP	FN	Precision	Recall	F-1 Score	Accuracy
Location	173	64	73	16	20	80%	76.2%	0.78	79.2%
Contact	132	48	57	14	13	77.4%	78.7%	0.78	79.5%
Credential	443	320	106	15	2	95.5%	99.4%	0.97	96.2%
Calendar	12	3	5	1	3	75%	50%	0.60	66.7%
Device/Card ID	373	161	180	22	10	88.0%	94.2%	0.91	91.4%
Phone Number	103	66	31	5	1	93.0%	98.5%	0.96	94.2%
Internet	1,009	606	319	52	32	92.1%	95.0%	0.94	91.7%
SMS	233	58	137	21	17	73.4%	77.3%	0.75	83.7%

Table 4: Comparison of *FlowCog* with other techniques

Variations	Keyword	Simple NLP	Similarity Model	Learning Model	Learning Model (Small training set)	Descriptions Only	Flow Contexts Only	All Semantics	*FlowCog*
Accuracy	73.5%	80.9%	79.3%	88.3%	65.5%	81.0%	82.5%	82.2%	90.2%

combining learning-based and learning-free, (ii) gradient boosting (GB) plus linear support vector machine (SVM) vs. other learning models, and (iii) NLP-based vs. keyword-based.

First, we would like to justify why *FlowCog* adopts a hybrid model that combines learning based and learning-free approaches. Table 4 shows the results of comparing the learning-free, learning-based, learning-based with a small training set, and hybrid (i.e., *FlowCog*). A pure learning-free approach, i.e., the similarity model in Table 4, has a bad overall accuracy, i.e., 79.3%, and that is why we need a learning-based approach. The overall accuracy of a learning-based approach is high, i.e., 88.3%, but such approach performs badly when the training set, e.g., these flows that leaking out Calendar via SMS, is small. Specifically, the accuracy, as shown in Table 4, is only 65.5% for such Calendar-to-SMS flows. Therefore, we choose a hybrid approach for the design of *FlowCog* in the end.

Second, we would like to justify the two learning algorithms, i.e., Gradient Boosting (GB) and linear Support Vector Machine (SVM), used in *FlowCog*'s learning-based approach. Specifically, we compare many different machine learning algorithms, including Logistic Regress (LR), Decision Tree (DT), Naïve Bayes (NB), linear Support Vector Machine (SVM) and Gradient Boosting (GB).

Table 5 shows the comparison results of different algorithms. The accuracies of efficient algorithms, such as LR, DT and NB, are all bad, i.e., below 80%. SVM and GB perform better with 81% and 84% respectively, but are still not satisfying. Therefore, we evaluated combinations of different algorithms in Rows 6–10 of Table 5. Among all the combinations that we evaluated, the combination of GB and SVM achieves the best results (93.1%). Note that one takeaway here is that classical efficient classification algorithms, e.g., LR, DT and NB, do not work well for our problem.

Lastly, we want to justify why we want to use NLP-based approach rather than a simple keyword-based one. Specifically, we implemented a keyword-based approach and compare it with *FlowCog*. Here is how the keyword-based approach, which measures the correlation between extracted semantics and target data flows, works. We manually generate 10 keywords for each category listed in Table 3. Each flow corresponds to two categories and thus has 20 keywords. Then, for each keyword, we get a list of synonyms using a Python library PyDictionary [3]. Next, we search each flow's keywords and their synonyms in their flow-related texts and descriptions. If we can find three matches, we will consider this flow as positive; otherwise negative.

Table 5: Accuracy of Different Learning Algorithms

Algorithm	Precision	Recall	Accuracy
Logistic Regression (LR)	84.2%	84.3%	81.9%
Decision Tree (DT)	73.8%	84.3%	73.8%
Naive Bayes (NB)	84.3%	83.3%	81.4%
Support Vector Machine (SVM)	86.8%	86.1%	84.5%
Gradient Boosting (GB)	84.2%	91.7%	85.3%
LR + DT	82.0%	84.5%	84.5%
LR + NB	84.5%	81.1%	80.6%
DT + SVM	85.3%	88.9%	86.0%
GB + NB	84.5%	88.9%	84.5%
GB + SVM	90.1%	93.1%	90.2%

Apart from the simple keyword-based approach, we also introduce a keyword-based approach with some simple NLP components. Specifically, we do not use keyword's synonyms, but parse the flow-related texts and descriptions using Stanford Parser [14]. We keep nouns and verbs as they usually contain a sentence's most information, do word stemming on remaining words, and discard the duplicate ones. Next, we compute the similarity score of this word list and the keyword list, using the Word2Vec similarity model discussed in Section A.4, and then make a classification decision based on the score.

We evaluate the keyword-based, keyword plus simple NLP, and *FlowCog* using the same testing set. The

Table 6: Accuracy in Extracting Flow-related Texts

App Type	# of Flows	# of Manually-found Text Blocks	# of Text Blocks Found by *FlowCog*	Accuracy
Benign	27	288	273	94.5%
Malicious	41	337	331	98.3%

Table 7: Accuracy in Extracting Flow-related Non-text Informative Elements

Type	# of Items in Total	# of Items solved by *FlowCog*	Accuracy
Image with Texts	30	27	90.0%
Image without Texts	23	23	100%
Non-image Views	2	2	100%

keyword-based approach performs the worst, only with 73.5% accuracy. The keyword-plus-simple-NLP approach is better than the pure keyword-based, achieving 80.9%. Note that this is also better than our similarity classifier alone with 79.3% accuracy. We do not adopt any keyword-based approaches in *FlowCog* because many manual works are involved and we want a fully automated approach.

5.5 RQ4: Effectiveness of Contexts Extraction

In this experiment, we study the accuracy of *FlowCog* in extracting flow contexts. Here is how we obtain the ground truth. We manually inspect 68 flows, i.e., these from ten benign apps in Google Play and ten malicious apps in Drebin dataset. In particular, we first instrument FlowDroid to display the detailed information of each flow, including the data path and call path, so that we know how to trigger the information flow. Then, we install and play with each app directly to trigger the information flow and record all the semantics that we see during the triggering process. Next, we decompile the apps using apktool [8] to find the classes that each statements in call path resides and map the semantics that we see to the corresponding text blocks or non-text items in the apps. These text or non-text resources are the ground truth used in this subsection.

Now let us look at the results. Table 6 shows *FlowCog*'s accuracy in extracting text-related contexts. In particular, *FlowCog* can extract 94.5% of flow-related texts from benign apps, and 98.3% of flow-related texts from malicious apps. We do not find any false positives, i.e., texts extracted by *FlowCog* are all related to the views.

Here are two reasons that *FlowCog* fails to extract some of the texts. First, three of the failed scenarios are caused by encoding issues of our implementation: some texts can be correctly rendered during our dynamic evaluation, but turn out to be garbled when extracted by *FlowCog*. Note that this is a minor implementation issue in converting texts in different encodings. *FlowCog* does support multiple languages: before feeding texts to classifiers, if any texts are not recognized as English, *FlowCog* will use a Python library called mtranslate [2] to translate them into English.

Second, the remaining 18 texts that *FlowCog* fails to extract are caused by the limitations of static value analysis: completely solving value analysis is still a fundamental challenges suffered by all static analysis tools. *FlowCog* adopts a bunch of heuristic rules to try our best to resolve those non-constant string values, but there are still 7 cases that we cannot resolve. Moreover, we also find 11 dynamic texts: the texts are dynamically loaded and cannot be found in the app's package. Static analysis cannot solve dynamically-loaded texts and the dynamic analysis tool that we use, i.e., AppsPlayground, does not trigger this specific code branch. Fortunately, most of dynamic texts have default values, which can be discovered by *FlowCog* and are usually sufficiently informative. For example, one gaming app will display various promotion texts during loading. Its default string value is "Now loading", which is sufficient to let user know that the app is using Internet.

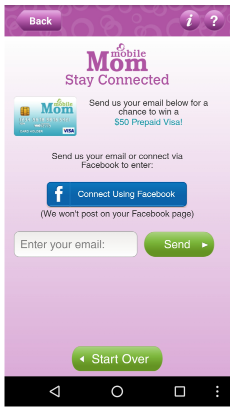
Next, Table 7 shows the accuracy of *FlowCog* in extracting information from informative non-text items: (i) images with texts, (ii) images without texts, such as mail icons, and (iii) non-image fragments, such as ads and maps. *FlowCog* can successfully extract 27 out of 30 texts embedded in images through OCR tool. The rest three images' texts are extracted as garbled texts. As for non-text images, there are 23 such images are informative to users. Google Images can successfully extract all of their semantic meanings.

For non-image views, we have seen many ad fragments, but we do not consider them as informative. Some ad library will send user's location to Internet for user targeting. However, we believe most users do not expect such location-leaking activities and thus we classify such flows as negative, unless other informative texts are given. We also see two map fragments in this experiment, which *FlowCog* can recognize.

6 Case Study

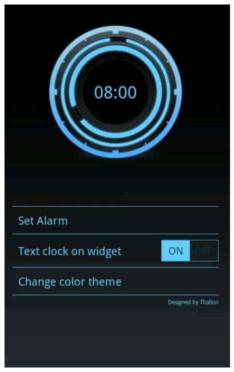
In this section, we perform a case study on a variety of data flows in different types of apps and discuss whether the app provides enough semantics for the flow, i.e. classified as positive or negative by *FlowCog*.

- **Positive and negative flows in the same app.** Due Date Calculator, shown in Figure 6a, is an app that allows a mother or mother-to-be to calculate her due date of an incoming baby. This app contains two flows, both from the database to the Internet. One flow is sending the email address of the user to the Internet, and the other is sending URLs in another database to

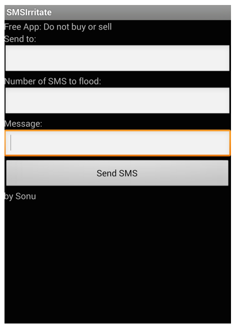

Back
mobile Mom
Stay Connected
Send us your email below for a chance to win a $50 Prepaid Visa!
Send us your email or connect via Facebook to enter:
Connect Using Facebook
(We won't post on your Facebook page)
Enter your email:
Send
Start Over

(a) Due Date Calculator – Mobile Mom

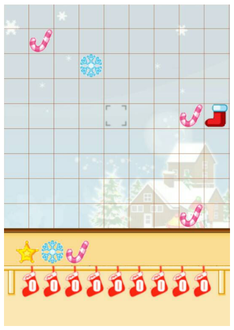
(b) Home of Ocarina



(c) Digital Clock Disc



(d) SMS Irrirate



(e) Merry Christmas

Figure 6: Screenshots of Different Apps in the Case Study

the Internet. *FlowCog* classifies the former as positive as flow contexts like "Send" and "Email Address" are available to the user, but the latter as negative due to lack of flow contexts. In fact, our manual inspection reveals that the database belongs to a third party library called Urban Airship, which is used to deliver third-party ads. The app user has no knowledge of such information leak. Note that existing app-level semantics correlation tools will not be able to differentiate such two flows, because they will ask for the same permissions.

- **A positive flow but not mentioned in the app description.** Home of Ocarina, shown in Figure 6b, is an official app of a company. This app contains a flow that leaks out users' geo-location. Interestingly, the app description only introduces some background information of the company, i.e., nothing related to geo-location. This flow is positive because the app allows a user to navigate to the Ocarina headquarter when she clicks the "Map" Button in the app. *FlowCog* can successfully extract flow contexts, such as "location of Home of Ocarina" and a Google map fragment, thus classifying the flow as positive. Note that this example is an good illustration of why we need flow contexts in addition to app descriptions.

- **A negative flow in a benign app.** Digital Clock Disc Widget (pl.thalion.mobile.holo.digitalclock) in Figure 6c is a benign app with a negative flow. Specifically, the app leaks out users' geo-location as well as the device ID to the Internet in an *onCreate* lifecycle callback. The app's description only shows how to add this clock widget to users' home screen, and the GUI of the app is about the clock only. That is, although the app sends out users' geo-location and device ID, no flow contexts are provided in the app. *FlowCog* marks this flow as negative because *FlowCog* only extracts "Set Alarm", "Text clock on Widget", "Change Color Theme", "–:–", "ON", "OFF" and "Designed by Thalion" from the app for the flow. None of the aforementioned texts are related to geo-location or device ID, and thus *FlowCog* cannot correlate the flow with the texts.

- **A positive flow in a malicious app.** SMS Irritate, shown in Figure 6d, is a malicious app from Drebin dataset [11, 25] with a positive flow leaking out user-specified information via short message. The purpose of this app is to send a large amount of user-specified messages to a designated phone number repeated and "irritate" the recipient. Although this is a malicious app, the flow is positive because the user of the app will understand that the app is used to send out messages. *FlowCog* will also mark the specific flow as positive, because *FlowCog* can successfully extract all the aforementioned texts, such as "Send to" and "Number of SMS to flood".

- **A negative flow in a malicious app.** Merry Christmas is another malicious app from Drebin dataset, which sends out users' information without their knowledge. Specifically, this app is a trojan, which pretends to be a gaming app, but hijacks the user's phone and leaks out confidential data while the user is playing the game. Figure 6e shows the interface of the trojan app. This malicious app has many information flows, including sending users' phone number, contacts, sim serial number and device ID to the Internet. *FlowCog* mark all the information flows in this app as negative, because no semantics are provided to justify these flows. Specifically, *FlowCog* successfully finds that all these flows are triggered by an *onCreate*() callback of an activity in the app and then extract semantics, which only include gaming tips, such as "Move the box to the target empty position ...", and app control information, such as "Are you sure you would like to exit?".

7 Discussions

First, we discuss the value analysis performed in *FlowCog*. We are aware that value analysis is a traditionally hard problem and cannot be solved solely by static analysis. *FlowCog* is able to resolve most, i.e., 95%, values for view IDs and strings, because these values are mostly static and pre-defined in Android apps. Even if they are defined dynamically in a rare case, *FlowCog* also relies on an optional dynamic analysis component to resolve the values.

Second, we discuss how clickjacking attacks, or in general UI redress attacks, influence our results. Simply put, these attacks are out of scope of the paper—all the information flows have already been given permissions in Android apps and thus the apps do not need a UI redress attack to fool the user to click something. More importantly, because *FlowCog* only identifies views that are related to a specific flow, other invisible views above or below are skipped by *FlowCog* and not considered in the semantics extraction stage.

Lastly, we talk about native code or JavaScript code in Android apps. FlowDroid does not support such non-Java code and thus *FlowCog* cannot deal with information flows related to native code or WebView-based JavaScript code either. We believe that *FlowCog* can be integrated with any future work that considers non-Java code, because semantics of Android apps are mostly provided in Java code.

8 Related Work

We discuss related works that apply either programming analysis or natural language processing on Android apps.

First, many works aim to detect information flows of Android apps [12, 15, 24, 30, 33]. FlowDroid [12] is a static precise taint analysis systems based on Soot framework. It is context-, flow-, field- and object-sensitive while still very efficient: FlowDroid transforms taint analysis's information flow problem into an IFDS problem, and then uses an efficient IFDS solver to find the solution. FlowDroid does not support inter-component analysis. To address this limitation, static analysis systems Amandroid [33], DroidSafe [18] and IccTA [24] are proposed to provide Android inter-component taint analysis. In addition to static analysis, dynamic analysis systems are also proposed to detect Android information flows. TaintDroid [15] conducts taint analysis dynamically by proposing a customized Android framework. Uranine [30], on the other hand, detects information leakage by instrumenting app without modifying the operating system. EdgeMiner [13] is an approach that detects implicit control flow transitions in the Android framework but does not analyze Android apps directly. None of these works attempt to infer whether an Android app provides sufficient semantics for information flows. That said, *FlowCog* can work with any such systems to determine whether enough semantics is provided.

Second, Android app's execution context is an important indicator to analyze app's behaviors. Several works are proposed to detect malicious Android apps based on execution contexts. AppContex [34] finds the contexts related to a set of suspicious actions, and then classifies the app as benign or malicious according to these actions as well as their corresponding behaviors. Similarly, TriggerScope [17] identifies narrow conditional statements, called triggers, and infers possible suspicious actions based on these triggers. DroidSift [36] classifies Android malware using weighted contextual API dependency graphs. As a comparison, *FlowCog* goes beyond app's execution contexts, i.e., activation events and guarding conditions, to find Android views and extract semantics related to these views.

Third, NLP techniques are also used in Android privacy. WHYPER [27] is the first work that aims to bridge the gap between semantics and behaviors of Android apps by using NLP techniques. Specifically, it extracts semantics from app's descriptions and API documents, and then determines whether the descriptions justify the usage of certain permissions. Another research work, AutoCog [28], tried to solve a similar problem with NLP on descriptions but used a learning-based approach using Android app's descriptions but not API documents. CHABADA [19] also extracts semantics from an app's descriptions, and then determines whether the app's API usages are consistent with the extracted semantics. Zimmeck et al. [37] propose another NLP system that extracts the semantics from app's privacy requirements and predicts whether an app is compliant with its privacy requirement. Apart from Android, NLP techniques have also been used in IoT devices to study privacy correlations [31]. AsDroid [20] correlates the stealthy behaviors of Android apps, such as a malware, with app's descriptions. DescribeMe [35] generates security-centric descriptions for Android Apps. As a comparison, *FlowCog* is the first system that analyze the correlation between information flows and the semantics—*FlowCog* faces additional challenges such as extracting flow-specific semantics.

9 Conclusion

Prior works correlating app behaviors and semantics are coarse-grained, i.e., on the app-level, which cannot provide insights for fine-grained information flow. Specifically, prior works cannot differentiate two flows, one with sufficient semantics provided in the GUI, i.e., available to the app users, and the other hiding secretly in the background.

In this paper, we propose an automatic, flow-level semantics extraction and inference system, called *FlowCog*. Given an information flow, *FlowCog* can extract all the related semantics, such as texts and images, in the app via a mostly static approach with an optional dynamic component. Then, *FlowCog* adopts natural language processing (NLP) techniques to infer whether the app provide sufficient semantics for users to understand the privacy risks, i.e., the information flow. We implement an open-source version of *FlowCog* with $\sim$16,500 lines of code available at `https://github.com/SocietyMaster/FlowCog`. Our evaluation results show that *FlowCog* can achieve a precision of 90.1% and a recall of 93.1%.

10 Acknowledgement

We would like to thank anonymous reviewers for their helpful comments and feedback. This work was supported in part by National Science Foundation (NSF) grants CNS-15-63843 and CNS-14-08790 and U.S. Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-15-C-7561. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF.

References

[1] Beautiful soup documentation. https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

[2] Mtranslate: A simple api for google translate. https://github.com/mouuff/mtranslate.

[3] Pydictionary: A "real" dictionary module for python. https://pypi.python.org/pypi/PyDictionary/1.3.4.

[4] Python interface to stanford core nlp tools. https://github.com/dasmith/stanford-corenlp-python.

[5] Python-tesseract: a python wrapper for google's tesseract-ocr. https://pypi.python.org/pypi/pytesseract.

[6] Scikit-learn: Machine learning in python. http://scikit-learn.org/stable/.

[7] Term frequency?inverse document frequency. https://en.wikipedia.org/wiki/Tf-idf.

[8] A tool for reverse engineering android apk files. https://ibotpeaches.github.io/Apktool/.

[9] Whyper site. https://sites.google.com/site/whypermission/.

[10] [wikipedia] bag of words model. https://en.wikipedia.org/wiki/Bag-of-words_model.

[11] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *NDSS*, 2014.

[12] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[13] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "Edgeminer: Automatically detecting implicit control flow transitions through the android framework," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[14] D. Chen and C. D. Manning, "A fast and accurate dependency parser using neural networks." in *Emnlp*, 2014, pp. 740–750.

[15] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 5, 2014.

[16] J. R. Finkel, T. Grenager, and C. Manning, "Incorporating non-local information into information extraction systems by gibbs sampling," in *Proceedings of the 43rd annual meeting on association for computational linguistics*. Association for Computational Linguistics, 2005, pp. 363–370.

[17] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "TriggerScope: Towards Detecting Logic Bombs in Android Apps," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, San Jose, CA, May 2016.

[18] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe." in *NDSS*. Citeseer, 2015.

[19] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 1025–1035.

[20] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 1036–1046.

[21] D. Klein and C. D. Manning, "Accurate unlexicalized parsing," in *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*. Association for Computational Linguistics, 2003, pp. 423–430.

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

[23] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for java program analysis: a retrospective," in *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, vol. 15, 2011, p. 35.

[24] 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 *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 280–291.

[25] S. Michael, E. Florian, C. F. Felix, and J. Hoffmann, "Mobilesandbox: looking deeper into android applications," in *Proceedings of the 28th International ACM Symposium on Applied Computing (SAC)*.

[26] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[27] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "Whyper: Towards automating risk assessment of mobile applications." in *USENIX security*, vol. 13, no. 20, 2013.

[28] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in android applications," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1354–1365.

[29] V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: automatic security analysis of smartphone applications," in *Proceedings of the third ACM conference on Data and application security and privacy (CODASPY)*. ACM, 2013, pp. 209–220.

[30] V. Rastogi, Z. Qu, J. McClurg, Y. Cao, and Y. Chen, "Uranine: Real-time privacy leakage monitoring without system modification for android," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2015, pp. 256–276.

[31] Y. Tian, N. Zhang, Y.-H. Lin, X. Wang, B. Ur, X. Guo, and P. Tague, "Smartauth: User-centered authorization for the internet of things," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 361–378.

[32] R. Vallee-Rai and L. J. Hendren, "Jimple: Simplifying java bytecode for analyses and transformations," 1998.

[33] 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 *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1329–1341.

[34] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "Appcontext: Differentiating malicious and benign mobile app behaviors using context," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015*, 2015.

[35] M. Zhang, Y. Duan, Q. Feng, and H. Yin, "Towards automatic generation of security-centric descriptions for android apps," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 518–529.

[36] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 1105–1116.

[37] S. Zimmeck, Z. Wang, L. Zou, R. Iyengar, B. Liu, F. Schaub, S. Wilson, N. Sadeh, S. M. Bellovin, and J. Reidenberg, "Automated analysis of privacy requirements for mobile apps," in *2016 AAAI Fall Symposium Series*, 2016.

A Context and Flow Correlation

A.1 NLP Module

FlowCog's NLP Module has four steps: preprocessing, parsing, grammar analysis, and post-processing. The first step separates raw texts into a list of sentences or phrases, and removes useless symbols; the second parses each sentence or phrase into a so-called grammar hierarchical tree by Stanford parser; the third converts each sentence or phrase to a list of action-resource pairs; and the last one processes the generated action-resource pairs. Here are the details.

First, *FlowCog*'s NLP module preprocesses all the raw input texts by annotating special nouns, such as email, abbreviation, IP address and ellipsis, by regular expressions. Then, the NLP module splits the input texts into sentences or phrases by special characters, such as "." and ":". (A full list of such characters is also used by existing work [9].)

Second, *FlowCog* adopts Stanford Parser [21] to process each sentence or phrase produced in previous step into a grammar hierarchical tree by extracting Stanford-typed dependencies, or for short typed dependencies, and *Part of Speech (POS)* tags of the sentence or phrase. Let us use a real-world sentence seen commonly in Android apps as an example. The sentence, indicating that the user's contacts are sent to the cloud for backup, is that "Your contacts are being synced with cloud." The Stanford Parser breaks down the sentence into multiple triples, each of which contains the *name of the relation*, the *governor* and the *dependent*, and outputs a grammar hierarchical tree.

Third, *FlowCog* converts the grammar hierarchical tree into a list of action-resource pairs, i.e., preserving the verb phrase with governor-dependent relationship from the Stanford parser. Specifically, *FlowCog* extracts all the noun phrases in the leaf nodes of the hierarchical tree and records all the verb phrases from their ancestors—the verb and the noun phrases form into an action-resource pair. Note that if *FlowCog* finds possessive node, e.g., "Your", such node will also be included in the resource; in addition, if *FlowCog* cannot find a verb node, a "null" action will be used. For example, from the "contacts" node, *FlowCog* will produce <null, "Your contacts">.

Lastly, after extracting all action-resource pairs as described, *FlowCog* further processes the extracted pairs. Particularly, *FlowCog* performs the following steps: (*i*) removing stopwords without sufficient semantic information, such as "are" and "the", (*ii*) replacing names, such as people and location, with general names by Stanford Named Entity Recognizer [16], and (*iii*) normalizing and lemmatizing all words, e.g., converting all letters to lowercase and plural subjects to singular. Consider our prior example. *FlowCog* will finally generate the following two action-resource pairs, <null, "your contact"> and <"synced", "cloud">.

A.2 Resource Filter

Resource filter is a component that filters common, non-informative words in the context of Android API documents. Examples are like "Android" and "App", because they are universal in the context of Android APIs. This is how resource filter works in detail. The resource filter groups action-resource pairs from Android API documents based on the flow types, i.e., sources and sinks, and then extracts all the resource phrases from the pairs. If more than half of the groups contain the same resource phrase (excluding "null"), *FlowCog* will consider this resource as non-informative and filter such action-resource pairs. Note that we adopt such tactics because if one resource appears in the API documents of more than half flow types, the resource is considered ineffective in differentiating the semantics of flows and thus safe to be filtered.

A.3 Learning-based Classifier

We now introduce the first category of classifiers, i.e., the learning-based one. This classifier takes the previously-generated two lists of action-resource pairs as inputs, and outputs a result about whether they are correlated. Specifically, there are three steps here. (*i*) *FlowCog* converts action-resource pairs into numeric feature vectors, called vectorization. (*ii*) *FlowCog* relies on two machine learning models, namely support vector machine (SVM) and gradient boosting (GB), to classify the generated feature vector as a correlation score. (*iii*) *FlowCog* uses logistic regression to calculate a combined score.

First, *FlowCog* uses a variation of bag-of-words to convert action-resource pairs to a text vector, and then adopts term-frequency inverse document-frequency (TF-IDF) model to convert the text vector into a numeric one. Specifically, *FlowCog* adopts bag-of-words model that considers the word order, i.e., each word and each action-resource pair are all separate elements in the bag. For example, if *FlowCog* sees two action-resource pairs, <find, friend> and <remember, me>, the generated text vector is <find, friend, remember, me, find friend, remember me>. Then, *FlowCog* converts each element, or called term, in the text vector to its TF-IDF value. The TF-IDF value for for each element is calculated as shown in Equation 1.

$$tfidf(t,d) = tf(t,d) * idf(t) = \frac{1+N}{1+df(d,t)} \qquad (1)$$

where the parameter t refers to the target element, the parameter d refers to the text vector, tf is the element's frequency, i.e., the number of times a term occurs in a given word pair list, idf is the element's inverse document-frequency, N is the number of text lists in our training set, and $df(dt)$ returns the number of text lists that contain the target element t. After calculating the numeric feature vector, *FlowCog* normalizes the vector using Euclidean norm and converts the vector to a sparse one for better accuracy and efficiency.

Second, *FlowCog* uses two classifiers, namely Gradient Boosting (GB) and Support Vector Machine (SVM), to predict a correlation score based the numeric feature vector outputted from the previous step. The adopted GB defines differentiable loss function and uses gradient descent approach to minimize the loss function in an iterative approach. in each iteration, *FlowCog* adds a new decision tree so the loss function on overall model will be decreased. The algorithm stops when the number of trees achieve a threshold, or the loss reaches an acceptable level, or the loss can no longer be decreased. Meanwhile, *FlowCog* adopts linear SVM in soft-margin version, which allows some points to be misclassified but each instance will impose a penalty to the target function.

Lastly, *FlowCog* relies on Linear Regression to combine results from GB and SVM into a single result that lies in between zero and one, where one means correlated and zero not.

A.4 Similarity Classifier

In addition to the learning-based classifier, *FlowCog* also has a learning-free classifier, called similarity classifier. Note that the terminology, learning-free, means that *FlowCog* does not require any training data from our dataset, i.e., anything from Android apps. Still, the model used in this classifier, namely Word2Vec, needs to be pre-trained from Wikipedia Corpus. Now let us discuss the details about how we use Word2Vec and calculate the similarity score.

First, we give some backgrounds about the word embedding model used in Word2Vec, the state-of-the-art and arguably the most popular predictive model to learn word embedding from raw texts. Traditionally, natural language processing encodes each word as discrete atomic symbols. For example, word "contact" is represented as "id171" and "connection" is represented as "id28". Such encoding scheme itself cannot provide information about the relationship between any two words. Assuming another word "rocket" is represented as "id211", we cannot conclude that "contact" is more related to "connection" than "rocket" based on their encodings. For comparison, word embedding encodes each word as a vector (e.g., $\vec{v}_{contact}$) and semantically similar words are mapped to nearby points in the continuous vector space. Therefore, we can calculate the similarity of any two words on their word embedding representation directly. For example, *cosine* function, which will be defined later in this section, is frequently used as a measure of similarity, so $cos(\vec{v}_{contact}, \vec{v}_{connection})$ larger than $cos(\vec{v}_{contact}, \vec{v}_{rocket})$ means the word "contact" is more related to "connection" than "rocket" in the specific model. Moreover, other operations on vector are also meaningful. Intuitively, if word A is related to either word B or word C, it is related to the word represented as $\vec{v}_B + \vec{v}_C$.

Second, we introduce how we use the Word2Vec model trained from Wikipedia corpus. *FlowCog* converts each action-resource pair to a vector via the vocabulary-vector mapping provided by Word2Vec. Specifically, *FlowCog* finds two vectors associated with action and resource separately, and adds these two vectors together as the final result. Note that there are two special scenarios. A "null" action will map to an zero vector, and if the resource contains more than one word, *FlowCog* will find the vector for each word and add them together.

Third, *FlowCog* calculates the similarity score between two vector lists corresponding to Android API documents and texts extracted by the Android app. In particular, *FlowCog* adopt cosine similarity as defined in Equation 2.

$$Similarity(List_a, List_b) = \sum_{i=1}^{M} \sum_{j=1}^{N} w_{ij} \cdot h(s_{ij}) \cdot s_{ij} \tag{2}$$

where M equals $sizeof(List_a)$, N equals $sizeof(List_b)$, and s_{ij} is $Similarity(\vec{v}_i, \vec{v}_j)$, the similarity score of two vectors as defined in Equation 3.

$$s_{ij} = Similarity(\vec{v}_i, \vec{v}_j) = cos(\vec{v}_i, \vec{v}_j) = \frac{\vec{v}_i \cdot \vec{v}_j}{\|\vec{v}_i\| \cdot \|\vec{v}_j\|} \tag{3}$$

Lastly, *FlowCog* needs to normalize the calculated similarity score, because the size of the vector list could affect the score. Specifically, we define an activation function $h(x)$ in Equation 4 to filter certain unrelated vector pairs when their contribution is small, and an exponentially decreasing weight function $w(x)$ in Equation 5 to reduce the effect of long list. Here is the definition of the activation function with an activation threshold as $threshold$.

$$h(x) = \begin{cases} 0, & x < threshold \\ x, & otherwise \end{cases} \tag{4}$$

We also define a weight function in Equation 5 with a decreasing factor as μ. The function assigns highest weight to the most related vector pairs (i.e., whose vector similarity scores are highest), second-highest weight to the second-most related pairs, and so on. So the most related texts contribute the most to the overall similarity scores.

$$w_{ij} = w(s_{ij} \cdot h(s_{ij})) = \mu^k, (0 < \mu < 1) \tag{5}$$

where k is kth element in $desc\_sorted(\{x | s_{ij} \cdot h(s_{ij}), i\varepsilon M, j\varepsilon N\})$. Note that both the activation threshold and decreasing factor are obtained empirically during our experiment. In practice, we choose 0.6 and 0.7 respectively for these two parameters.