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

Xuechao Du, Xiang Pan, Yinzhi Cao, *Member, IEEE,* Boyuan He, Gan Fang, Yan Chen, *Fellow, IEEE,* and Daigang Xu

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. They can only identify whether an app, as a whole, should request special permission but cannot answer whether a specific app behavior under a particular runtime context, such as information flow, is correctly justified. We propose *FlowCog*, an automated system to extract semantics related to information flows and correlate such semantics with given information flows to address these issues. Particularly, *FlowCog* statically finds all the Android views 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 natural language processing and deep learning approaches to infer whether the extracted semantics correlate with the given flow. *FlowCog* is open-source and available at https://github.com/xcdu/FlowCog. Our evaluation shows that *FlowCog* can achieve an accuracy rate of 95.4% and an F_1 score of 0.953.

Index Terms—Android, Information Leakage, Semantic Extraction, Natural Language Processing

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 phone book to ease the dialing. While these examples provide legitimate usages of private information, some apps may also misuse such information [1]–[4], such as stealing users' call history without their knowledge.

That said, an app needs to justify access to users' private information with sufficient semantics available to users. For example, a weather app will clearly state that it provides local weather conditions to make user understand its access to location information. Existing researchers have already started to study the semantics of an app's behaviors. For instance, CHABADA [5], Whyper [6], and AutoCog [7] try to correlate the app's description, such as these in Google Play, with the permissions that the app asks.

However, existing approaches [5]–[8] 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 a specific context. For example, an app may have two data flows[1] [9]–[14] accessing private information. The first one provides customized service with user's knowledge, e.g., a pop-up window, but the other hiding secretly in the background and sending information to the Internet without the user's knowledge. The former is legitimate with sufficient semantics, which we call *positive* in the paper, but the latter is *negative*.

This paper proposes an automated, flow-level system called *FlowCog* to extract and analyze semantics for each Android app's information flow.

FlowCog is fine-grained because it extracts flow-specific semantics called *context*, e.g., the 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 critical insight of *FlowCog* is that apps embed flow contexts 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 flow context will be in the view that has the submit button. Particularly, *FlowCog* performs a static analysis that connects UI views, such as button and checkbox, of Android apps with given information flows via control and data dependencies. *FlowCog* extracts flow contexts, e.g., texts and images, embedded in UI views via a mostly static approach with an optional dynamic component.

- *X. Du is with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China. Email: xcdu@zju.edu.cn.*
- *X. Pan is with the Google Inc., Email: xiangpan2011@gmail.com.*
- *Y. Cao is with the Department of Computer Science, Johns Hopkins University, Baltimore MD 20218, USA. Email: yinzhi.cao@jhu.edu.*
- *B. He and Y. Chen are with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208 USA. Emails: bhe@northwestern.edu, ychen@northwestern.edu.*
- *G. Fang is with Palo Alto Networks. Email: gfang@paloaltonetworks.com.*
- *D. Xu is with ZTE CORPORATION. Email: xu.daigang1@zte.com.cn.*

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

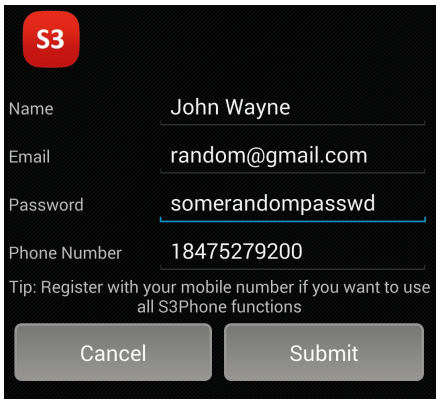


Fig. 1: Registration Interface of S3 World Phone App

Once *FlowCog* extracts flow contexts, it distills texts from images via image recognition and then analyzes texts, including these extracted from images using an NLP module. Lastly, *FlowCog* adopts 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 negative.

We have implemented a prototype of *FlowCog*. Our evaluation against 52,221 flows extracted based on FlowDroid [9] framework shows that *FlowCog* has a 96.4% precision, a 94.1% recall, and a 95.4% accuracy.

We claim that a conference version of the paper has been published at the 27th USENIX Security Symposium [15]. In this paper, we improve on the shortcomings of the original similarity approach and propose a new model that allows *FlowCog* to determine unauthorized flows accurately. Based on the original Drebin dataset and Android apps from the Google Play Store, we expand our focus to third-party Chinese Android app marketplaces. Our experiments show that the new method provides a significant improvement in prediction accuracy over the old version.

Besides, *FlowCog* is open source and available at the following repository: https://github.com/xcdu/FlowCog.

2 Overview

This section gives a running example, called the S3 World Phone app (called S3 app for short), allowing users to make phone calls worldwide. The S3 app sends a user-provided phone number to its server after the user sees a registration page shown in Figure 1 and presses the "Submit" button. This flow is from the phone number to the Internet, and it 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.

FlowCog 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, as shown in Figure 2, such a process can be broken down into two stages or six complex components. According to Figure 2, stage (a) dependency analysis includes components (i) flow analyzer, (ii) view content analyzer, (iii) special statement discovery engine, and(iv) view dependency explorer to resolve data flows and explore the view contexts related to the data flows. Stage (b) semantic correlation includes components (v) semantic correlation module and (vi) annotation metric module, which function on semantically correlating the flows and view context (i.e., semantics), then automatically provide judgment on whether app provides enough semantics to users.

Flow analyzer finds information flows of an Android app, while view content analyzer helps identify the content from the app's layouts. Special statement discovery engine locates special statements called *activation event* and *guarding condition* via control dependency and associated views (called view dependency) for each information flow. View dependency explorer recognizes and extracts contexts, e.g., texts and images, from those two special statements via data dependency. The semantic correlation module determines the correlation between the flow and the contexts via Natural Language Processing (NLP) technique, and the annotation metric module helps classify the sensitive semantic context and annotate the flow and view context.

Now we use our running example to illustrate how the process works. First, *FlowCog* will rely on the existing framework, such as FlowDroid, to find the Android app's information flow. 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*, 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 the *activation event* because an onClick event activates the $performRegistration$ method in Block 6. The second statement in Block 2 shows an example of the *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 to 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. *FlowCog* finds additional special statements, 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* 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

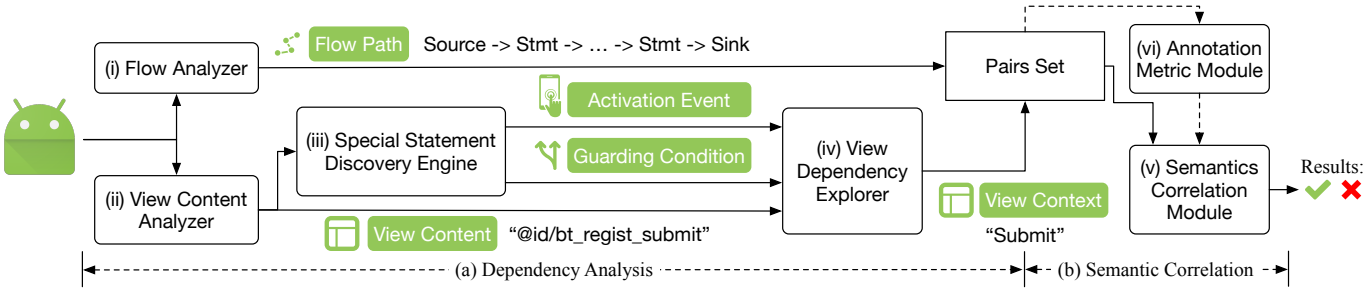


Fig. 2: *FlowCog* Architecture

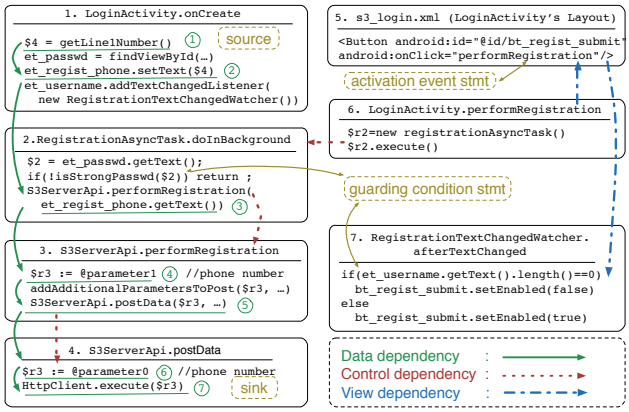


Fig. 3: Simplified Code Blocks of S3 World Phone App

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, removes stop words corpus based on the corpus, and distills phrases through constituency parsing [16]. For instance, the texts mentioned above are formatted into "submit", "password" and "tips register mobile number."

FlowCog feeds the flow contexts and flows into the trained classifier to predict the correlation between them. According to our correlation, the flow is considered as positive because "tips register mobile number" is related to the source while "submit" and "password" are related to the sink semantically.

3 Dependency Analysis

This section presents the details of each component of *FlowCog*'s architecture in Figure 2 Stage (a). Information flow analysis, i.e., step (i) in Figure 2, is skipped because we use existing taint analysis tool, FlowDroid [9]. We first present the special statement discovery engine in Section 3.1, which finds both activation events and guarding

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

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

statements. Then we present view dependency explorer in Section 3.2. Later, we show how to extract semantics from views and other places in Section 3.3. 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 for 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. The semantics associated with these two special statements will influence users' decisions and perceptions 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 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 given data flow call path 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*. Let us take UI events as an instance. *FlowCog* extracts callback methods from both codes and layout files.

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
```

Specifically, *FlowCog* parses the app's codes to identify all those event listener registration statements (e.g., setOnClickListener(...)) and then gets the callbacks by extracting the name of the 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 $onCreate$.

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 if 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 statement 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 if 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 data flow's execution. For example, if one branch of an if statement allows the data flow but another terminates the flow, we consider such if statement as a guarding condition—both if statements in Blocks 2 and 7 in Figure 3 are such examples. We now formally define the guarding condition in Definition 2.

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 directly connected with the data flow in the iCFG; and then, in the second phase, *FlowCog* identifies guarding conditions 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* searches 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 graph (Line 13–14).

We then discuss the search algorithm mentioned in the 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 the 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 a child, the conditional statement is a guarding condition.

Next, *FlowCog* finds all the conditional statements that control the given data flow's activation event 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 to

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
```

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
```

extract semantics. We call such a relationship between views and the data flow *view dependency*. Specifically, we classify view dependencies into the following three categories.

- Data flow related. A view can be dependent on the given data flow directly. For example, if the data flow source 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 an 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 three scenarios mentioned earlier, i.e., the given data flow, its activation events, and its guarding conditions. Now, let us explain in detail 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 the source. Besides, *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 the source.

Second, *FlowCog* obtains all the sinks based on the dependency categories. Statements in the given data flow and guarding conditions are added directly to the sink list. *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 directional 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 have the dependency on a given data flow.

3.3.1 Flow Context Extraction from Views

There are two types of flow contexts: those from views that have dependencies on 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 on a given data flow, 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. Its text, i.e., the "submit" word, is the semantics about sending behavior. For another example, an "alert" Dialog view asking for a user's permission to share 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 the 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, 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 where the depended view locates. 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 and all new statements that create dynamic views. (iii) *FlowCog* extracts semantics from other views just as what it does for the target depended view.

3.3.2 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) [17] engine to extract obvious texts.

Third, *FlowCog* also adopts Google Image to analyze the topics of images extracted in the 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 images 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* relies on a manual-curated list to extract semantics. Let us take the Google Map as the instance. 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, when *FlowCog* finds this fragment in a layout file or related code, a "map" semantics will be added.

3.4 Inter-Component Communication Analysis for *FlowCog*

The Intent mechanism [18] allows Android to communicate crossing components. The Inter-Component Communication (ICC) also brings increasing complexity to *FlowCog* dependencies analysis. We modified S3 World Phone App in Figure 1 into an ICC case in Figure 6, and illustrate how *FlowCog* performs dependency analysis with ICC.

2. We present the source code in List 1 in Appendix.

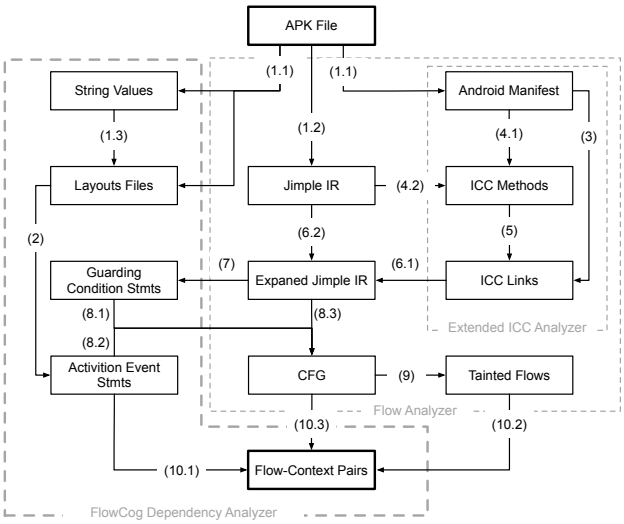


Fig. 5: The FlowCog Dependency Analysis Workflow for Inter-Component Communication.

Fig. 6: The *FlowCog*'s Dependency Analysis Details of Modified ICC S3 World Phone App Example.[2]

The Figure 5 prensents the entire workflow of *FlowCog*'s dependency analysis for ICC apps. Note that we use the FlowDroid [9] as flow analyzer and IC3 [19] as extended ICC analyzer.

When we feed the ICC APK file into the flow analyzer, the *FlowCog* will collect the string id definitions, extract the layout trees for every components, analyze the component relationships from Manifest file extracted, and decompile the APK file into Jimple in the Step 1.*. Then *FlowCog* will analyze the activition events statements as same as what *FlowCog* has done to the non-ICC apps in Step 2. The Step 3 to Step 6 are typical ICC taint analysis steps, and the details can be refered from [11]. After Step 6, the ICC links will expand the original controw flow graph generated by flow analyzer and extend edges, lifecycles and callbacks found in ICC analysis progress. Then, the *FlowCog* will collect the guarding condition statements from expanded IR in Step 7. In Step 8.*, the special statement discovery engine will help to locate those two kinds of statements in CFG. In Step 9, the flow analyzer will help to extract the tainted flows for

ICC apps. At last, *FlowCog* will gather all activation event statements in Step 10.1, tainted flow paths in Step 10.2, and search the reachable views on CFG in Step 10.3. The view dependency explorer will help to extract views related to the ICC tainted flows in Step 10.3.

We note that this research do not aims on any ICC taint analysis algorithm improvement and the *FlowCog* is compatible enough with the IC3. So we just leverage the existing IC3 framework in Extended ICC Analyzer in Figure 5 without any further modification.

A Case Study for *FlowCog*. According to [11], we modified the example in Figure 3 to explain the workflow in details. The Figure 6 illustrates the results from *FlowCog* in dependency analysis stage. we make *S3 World Phone App* to fetch phone number and send it in different activities through the *Intent*.

Unlike Figure 3, we have explicitly registered the listener for the button *bt_regist_submit* via *setOnClickListener* and the declared *OnClickListener* also explicitly constructs an Intent. At the same time, we receive this Intent in another Activity *RegActivity* and extract the password we need inside it. Then the password is sent out by *HttpClient.execute()* inside the same function *postData* as shown in Figure 3. *FlowCog* has found the tainted flow from the source *getLine1Number()* to the sink *HttpClient.execute()*.

By bytecode-rewriting from IC3, *FlowCog* expands the ICC edges from *LoginActivity*$\dot{}$*this*$\dot{}$*startActivity()* to *onStart()*, which is explicitly declared in *RegActivity*. Thus, the control flow graph connects the two activities *LoginActivity* and *RegActivity*. By locating the activition event statements and guarding conditions, we can resolve the statements of *findViewById()* backward through the data dependency and control dependency. Then we can find the related strings presented to the user through the string IDs or data flow analysis, and connect the semantics to the tainted flows.

3.5 View Content Analyzer

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% of statements belong to such category. The dynamic analysis works in three steps.

First, the dynamic analysis instruments the Android app by identifying all the text-setting statements and printing the values their parameters and 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 [20] to install the app on the 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. We set each app 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 and the statement's location will be printed out. After execution, these logs will be extracted and stored in a NoSql database. Each record's key is the app's name and the statement's location, while the value includes the texts associated with the corresponding statements' arguments. If *FlowCog* encounters a string argument whose value cannot be resolved during the static analysis, it will look up the database built from dynamic analysis.

4 Semantics Correlation

This section illustrates the details of semantics correlation stage in Figure 2 Stage (b). We first explain the problems that arise when we apply *FlowCog* to a larger dataset in Section 4.1. Then, we present the details of the semantic correlation module in Section 4.2. Finally, we introduce the annotation metric module and ground-truth-based annotation method in Section 4.3.

4.1 Problems in Data Increasing

The semantic correlation module helps determine the semantic connection between the *flow* and *flow context*[3] pairs. It mimics the user to determine if the user has been given enough semantic information to realize that data is being leaked out through the flow. However, as the number of data increases, the old semantic correlation model used in [15], shown in Figure 8, reveals its shortcomings.

First, the method of comparing flow and semantics used in [15] causes the similarity obtained from similarity classifier to tend towards its value domain boundary as the amount of context increases. It causes the similarity of positive and negative flow to be crowded into a narrow interval, which prevents the logistic regression from reaching a high accuracy.

Second, the step of mimicking the user's judgment of whether the semantics extracted along with the flow provides sufficient semantic information is very subjective. The supervised learning model requires a large amount of annotation data, and annotating data requires time to review flow. They lead to an inefficient annotation process and inconsistent annotation standards.

To solve the above problems, *FlowCog* proposes a feature method based on behavioral ontology. The feature method establishes a link between flow and semantics in both the semantic correlation module and the annotation metric module. We then determine the legitimacy of flow by using existing NLP techniques and deep learning text classification model, combined with the annotation set we built upon ground-truth via the annotation metric module.

4.2 Semantic Correlation

4.2.1 Behavioral Ontology

We use the so-called *behavioral ontology* to compare flow and flow context semantically. This ontology is used in our previous work [15], called Bag of Words Similarity Model (*BoWSiM*), shown in Figure 8, and then improved to extract valuable information on the pairs and to accommodate large amounts of data annotating.

3. We also use term *semantics* to refer to *flow text* or *flow context*.

Behavior is a pair of phrases consisting of *action* and *resource*. Action is a verb or verb phrase representing the action performed, while the resource is a noun phrase that represents the object of the action. For the example flow illustrated in Figure 3, *FlowCog* can extract "post data" from $S3ServerApi.postData$ in flow, and "register mobile number" from "Tips: with mobile number ...".

FlowCog can identify potentially sensitive information. By comparing flow's and semantics' behaviors, it can determine whether flow and semantics describe one or a class of behaviors simultaneously and semantically, i.e., whether sufficient semantics information is provided for the purpose. For the above example, the "register phone number" provides enough information to inform the "post data" behavior, which indicates it is a positive flow then.

4.2.2 Preprocessing and Semantic Expansion

As illustrated in Figure 7, the preprocessing module receives the flow and semantics pair extracted from the dependency analysis stage to eliminate the unnecessary characters. Then *FlowCog* splits the words that are joined together in flow and semantics. After that, *FlowCog* splits flow by statement and semantics by sentence. Named entity recognition and stemming of words are used to avoid interference from proper nouns and tense. Since we leverage the behavior ontology as the criterion for comparison, irrelevant actions and resources should be excluded in the preprocessing.

The semantic expansion module consists of three key components, as shown in Figure 7, behavior extraction, synonym expansion, and semantic alignment. The behavior extraction component extracts the pairs of actions and resources. Furthermore, *FlowCog* appends the pairs after the original flow or semantics. Synonym expansion is the component that addresses the domain-specific synonym similarity problem. It cooperates with the sensitivity level mechanism to attach possible synonyms to flow or semantics. For example, the contact list and address book usually refer to the same part in the Android system but do not appear together in the same flow. The additional synonyms increase co-occurrence and inform the model that they perform the same sensitive behavior or potentially threaten the same resource.

Linguistic alignment is to provide consistent support for multilingual contexts. Typically, developers tend to provide multilingual support for apps that are not in the English market. Often, characters belonging to different languages will appear simultaneously. For this mixture of characters, we translate the text into the same language to align the semantics so that the behavior comparison mechanism can function properly.

4.3 Annotation Metric

4.3.1 Sensitivity Level Mechanism

We adopt the so-called sensitivity level mechanism to improve the efficiency of annotating numerous flow-semantics pairs and reduce the annotating process's subjective. To clearly define when semantics provides enough information to the user, we define the relationship between the behaviors described by flow and semantics as follows.

First, each behavior has its sensitivity level, which is categorized by its relevance to permission. Sensitivity levels include: *restricted*, *noticed*, and *unrelated*. The restricted behavior indicates that it is significantly related to dangerous or signature permission. If it occurs in the flow or semantics, the same or similar semantic information needs to be present on the other side. The noticed behavior means that the behavior is related to the normal permission. If it occurs in the flow, the same or similar semantic information needs to be present in the semantics, but not vice versa. The unrelated behaviors should be ignored in both comparison and updating.

The basis for deciding whether flow and semantics are the same or similar is considered in two dimensions: action and resource. Behavior action cannot be directionally inconsistent, such as "post" and "fetch". *FlowCog* only requires resources to be semantically the same or similar for noticed behavior, provided there is no directional inconsistency. In the case of restricted behavior, both action and resources need to be semantically the same or similar. The semantic similarity is provided by the fine-tuned embedding model [21].

4.3.2 Constructing Annotation Criterion

We obtain the annotation of pairs by constructing annotation criteria and checking annotation in sequence. Constructing annotation criterion includes initializing and updating.

To obtain ground truth, we first need to extract the sensitivity level mapping table from the API archives. For the flow analyzer used in the dependency analysis of 2, all sources and sinks must be pre-defined. Their associated permission types and their sensitivity level are also accessible from the official documentation. We treat the source or sink as a flow consisting of one statement and the information describing those APIs as semantics. Then we can initialize a set of behavioral sensitivity levels from that. Simultaneously, we construct the embedding model and expand the vocabulary via Wiki Corpus [22], and then fine-tune the semantic distance of words in the embedding model via synonyms dictionary. Later, we continue to fine-tune the embedding model by using the pair set obtained from the dependency analysis in 2 and the annotated data set obtained from the *BoWSiM*. Ultimately *FlowCog* obtains a similarity estimation of behavior. We note that it is not the final prediction. Since the process of acquiring the pairs set may have a bias in the number of pairs. The method based on a given observation, such as what is used in *BoWSiM*, is not applicable here.

With this embedding model, we obtain a rough inference of the sensitivity of behavior. It allows us to convert the work of annotating the training set of thousands of flow pairs into labeling the behaviors with sensitivity levels. This kind of annotation criterion obtained is ground-truth based, and each subjective determination of flow is constrained within a single behavior and does not affect the annotating for different kinds of permission, making the impact of subjectivity minimized.

5 Implementation

Now we discuss the implementation of *FlowCog* in this section.

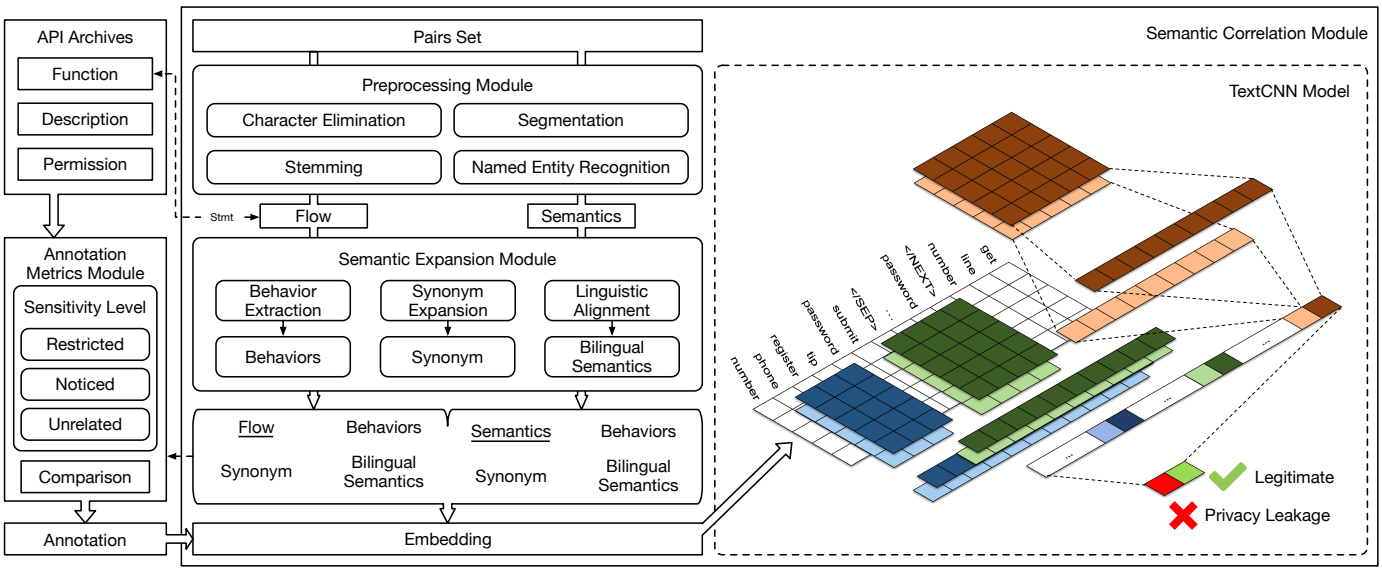


Fig. 7: *FlowCog*'s Semantic Correlation Module and Annotation Metric Module

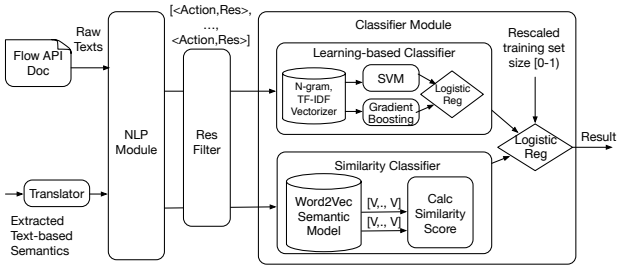


Fig. 8: Bag of Words Similarity Model

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) [23], a typed 3-address IR suitable for optimization and easy to understand. *FlowCog* uses Soot framework [24] 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 [25] to crawl API documents for methods associated with each flow. Then we use Stanford Parser Wrapper [26], 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 [27], a Python wrapper of Google Translate API, to translate non-English texts into English. *FlowCog* leverages Gensim [28] pre-trained Word2Vec model to embed texts into vectors. *FlowCog* uses Python's Tensorflow [29] and Scikit-learn library [30], which integrates all the machine learning modules we have used in our implementation and evaluation.

Lastly, we use apktool [31] 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 [32], 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 to automatically instrument apps and then manage and customize AppsPlayGround [20] to dynamically explore the instrumented apps.

6 Evaluation

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

- RQ1: How accurate is *FlowCog* in identifying positive and negative flows?
- RQ2: What's the performance in both Dependency Analysis Stages?
- RQ3: How effective is *FlowCog* in extracting flow contexts?
- RQ4: What's the insight of the classified flow context from *FlowCog*?
- RQ5: How does *FlowCog*'s classification algorithm compare with other alternatives, naïve approaches?

6.1 Experiment Setup, Dataset and Ground Truth

The composition of the evaluation dataset is incrementally growing as the version of *FlowCog* is iterated. Currently, the evaluation dataset consists of the following parts:

- Large-scale annotated dataset for semantic correlation accuracy evaluation and practical applications.
- Modified ICC-Bench dataset [33] for ICC performance evaluation. We added view components for each type of ICC types.
- A small-scaled annotated dataset randomly sampled for model selection.

The first large-scale annotated dataset consists of two parts: A small Android app set we formerly used in the [15] and evaluated by *BoWSiM*. And a large bilingual Android

app set we used in the latest version of *FlowCog* and evaluated by *TextCNN* model. The former small app set contains 4500 apps randomly crawled from Google Play and 1500 malicious ones randomly selected from Drebin dataset [34], [35]. Because we only parsed the English characters[4] for evaluation from this app set, we call it the *Monolingual App Set*.

The latter large bilingual Android appset not only includes the English Android apps from Google Play App market and Drebin malicious dataset, but also includes Chinese Android apps from the third-party Chinese Android app market and collected malicious Chinese Android app dataset of our other projects. This Android app set contains total 20000 apps, and we call this app set the *Bilingual App Set*.

For dependency analysis, we run experiments on a Ubuntu 16.04 server with Intel Xeon 2.8G, 16 cores CPU, and 64G memory. For semantic correlation, we perform our NLP experiments on a Ubuntu 16.04 server with Intel i7-7700k 4.5GHz, Nvidia GTX 1080Ti, and 32G memory.

For large-scale analysis, *FlowCog* uses FlowDroid [9] with default setting, i.e. flow-sensitive and context-sensitive, as the existing static analysis tool to extract tainted flows. Specifically, we use latest version of FlowDroid with the '-im' parameter and IC3 [19] to evaluation ICC dependency analysis on our Modified ICC-Bench dataset. We run FlowDroid on each app for 20 minutes and then terminate it if there are no 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 want 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.

For TextCNN model, the *FlowCog* is tuned with various value of hyperparameters. Finally, the semantic correlation *TextCNN* model sets *batch size* as 25, *dropout rate* as 0.5, *embedding dimension* as 200, *filter size* as {3, 4, 5}, *number of hidden unit* as 300, and *number of filters* as 32. The other parameters use the default value. Usually, we can reach the claimed accuracy within 30 minutes.

In the end, as to the *Monolingual App Set*, 1885 apps terminate successfully, and 947 of them (361 from Google Play and 586 from Drebin) generate 2342 flows in total. As to the *Bilingual App Set*, 7871 apps terminate successfully, and 4624 generates 52221 flows. We spent around three months to annotate all those flows manually based on our Annotation Metric Module.

6.2 RQ1: Precision, Recall and Accuracy

In this research question, we measure *FlowCog*'s *true positive* (TP), *true negative* (TN), *false positive* (FP), and *false negative* (FN) based on our manually annotated ground truth. We further calculate the *precision*, *recall* and *accuracy* from TP, TN, FP, and FN. Precision is defined as $TP/(TP+FP)$, recall as $(TP/(TP+FN)$, accuracy as $(TP+TN)/(TP+TN+FP+FN)$, and F_1 score as $2TP/(2TP+FP+FN)$.

As shown in Table 1, we conducted four rounds of experiments to demonstrate the *TextCNN* correlation model's performance and to compare it with the *BoWSiM*. For the former *Monolingual App Set*, we used 1242 of 2342 annotated flows as the test set, which is the same size as the test set we used in [15]. For the new *Bilingual App Set*, we used 10455 of 52221 annotated flows as the test set, which means the *Bilingual App Set*'s test rate is 20%.

The first row in Table 1 shows the result from [15], which we obtained 90.2% accuracy, 90.1% precision, and 93.1% recall. We replaced the correlation module from *BoWSiM* into *TextCNN*, and the accuracy, precision, and recall increase to 95.1%, 97.4%, and 94.0%, respectively. In the third row of Table 1, we feed the larger amount of data, *Bilingual App Set* into *BoWSiM*, as we explained in Section 4.1, the accuracy, precision, and recall rates drop to 72.0%, 74.2%, and 71.0%. Finally, when we use the *TextCNN* semantic correlation model to the *Bilingual App Set*, the accuracy, precision, and recall rates are 95.4%, 96.4%, and 94.1%.

The result demonstrates that the *BoWSiM* from [15] can have a performance drop with more flows and flow texts. However, the *TextCNN* improves the prediction performance and is robust in semantic correlation with multiple languages.

6.3 RQ2: Performance in Dependency Analysis

We selected 3 dimensions to evaluate the performance of the dependency analysis stage: runtime, Android application size, and the number of discovered flow-context pairs.

As shown in Figure 9, we cyclically compared such three dimensions on randomly sampled 1428 pairs from the pairs set of *Bilingual App Set*. The Figure 9a compares runtime in dependency analysis for each app with apk file size and number of found pairs. The Figure 9b compare the size with runtime and found pairs number. And Figure 9c compare the pairs number found in dependency analysis stage with the runtime and size.

We use such redundant comparison to intuitionally represent the distribution of each dimension, and the relationship between each two pairs of dimension.

Then we summarize on each dimension. In Figure 9a, the runtimes of processed Android apps are clustered below 300 seconds and among 700 to 1200 second. The number of pairs is more dispersed compared to the file size. In Figure 9b, the majority of the APK file size is below 100MB. Based on that distribution, we manually checked the entire file size of our entire *Bilingual App Set*, we found that most of the apps larger than 100MB can not reach to the result under 20 minutes. In Figure 9c, the majority of the pairs number is below 75. When the size is closer to the zero, the number of pairs is more sparse.

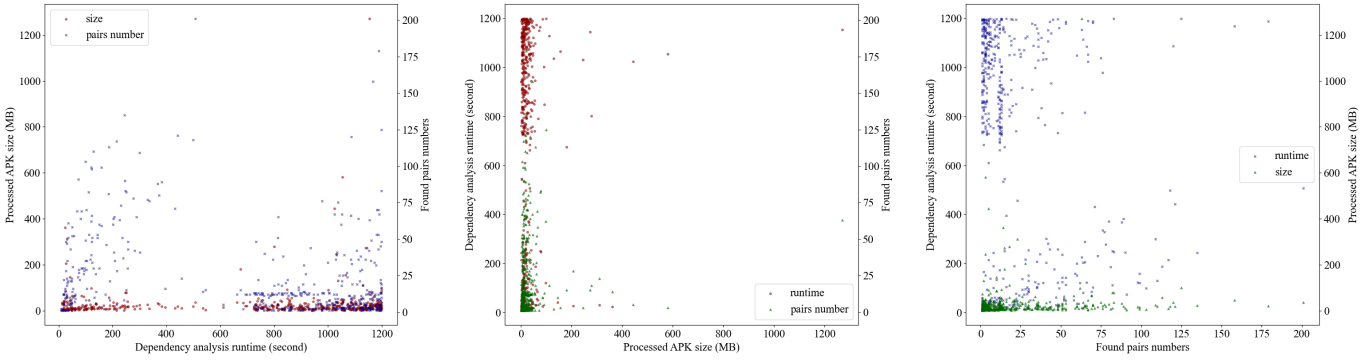
Alongside the runtime performance of dependency analysis, according to the the Table 4 and our manually checked results, the network access and credential access will extremely increase the number of control flow graph edges. It would leads to the challenge for terminating dependency analysis in acceptable time, as well as the challenges in user-awared tainted flow classification.

4. The English characters include the English letters, the symbolic, and the numeric characters used in the English context. The Chinese characters include the Chinese characters, the symbolic, and also numeric characters used in the Chinese context.

TABLE 1: Manually-annotated Ground Truth and Overall Performance of *FlowCog* against the Ground Truth

No.	Correlation Model	App Diversity	Total Pos.	Total Neg.	TP	TN	FP	FN	Precision	Recall	Accuracy	F_1 Score
1	*BoWSiM*	Monolingual(EN)	713	529	664	456	73	49	90.1%	93.1%	90.2%	0.916
2	*TextCNN*	Monolingual(EN)	713	529	670	511	18	43	97.4%	94.0%	95.1%	0.956
3	*BoWSiM*	Bilingual(EN+ZH)	5230	5215	3,883	3,538	1,677	1,347	69.8%	74.2%	71.0%	0.720
4	*TextCNN*	Bilingual(EN+ZH)	5136	5309	4,833	5,130	179	303	96.4%	94.1%	95.4%	0.953

*Monolingual app set only feeds English (EN) characters as input, while bilingual app set feeds both English and Chinese (ZH) characters as input.



(a) Dependency Analysis Runtime Comparison

(b) Processed Android Application Size Comparison

(c) Found Android Pairs Number Comparison

Fig. 9: Cycle Comparison Scatter Figures of *FlowCog* Dependency Analysis Stage.

TABLE 2: Accuracy in Extracting Flow-related Texts

App Type	#Flows*	#$T_{Manually}$†	#$T_{FlowCog}$‡	Accuracy
Benign	27	288	273	94.5%
Malicious	41	337	331	98.3%

* number of flows. † number of text blocks found mannually. ‡ number of text blocks found by *FlowCog*.

TABLE 3: 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%

6.4 RQ3: 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 the Drebin dataset. In particular, we first instrument FlowDroid to display the detailed information of each flow, including the data path and call path, to know how to trigger the information flow. Then, we install and play with each app to trigger the information flow and record all the semantics that we see during the triggering process. Next, we decompile the apps using apktool [31] to find the classes that each statement in the 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.

Table 2 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*.

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 challenge 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 dynamic texts have default values, which can be discovered by *FlowCog* and are usually sufficiently informative. For example, one gaming app will display various promotional texts during loading. Its default string value is "Now loading," which is sufficient to let the user know that the app is using the Internet.

Next, Table 3 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 the OCR tool. The rest three images' texts are extracted as garbled texts. As for non-text images, 23 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,

TABLE 4: Flow Classification Metrics by Permissions

Permission	Number	TP	TN	FP	FN	Precision	Recall	Accuracy	F_1 Score
Location	173	64	73	16	20	80%	76.2%	79.2%	0.78
Contact	132	48	57	14	13	77.4%	78.7%	79.5%	0.78
Credential	443	320	106	15	2	95.5%	99.4%	96.2%	0.97
Calendar	12	3	5	1	3	75%	50%	66.7%	0.60
Device/Card ID	373	161	180	22	10	88.0%	94.2%	91.4%	0.91
Phone Number	103	66	31	5	1	93.0%	98.5%	94.2%	0.96
Internet	1,009	606	319	52	32	92.1%	95.0%	91.7%	0.94
SMS	233	58	137	21	17	73.4%	77.3%	83.7%	0.75

but we do not consider them as informative. Some advertise libraries will send the user's location to the 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.

At last, for contexts extraction on ICC apps, the *FlowCog* passes all test cases in ICC-Bench. Note that for each test case, we just simply registered the views and view strings. Then we added a view box by *findViewById* in the Intent broadcasting activity. Then we build a data dependency crossing the ICC tainted flow by fetching the string value of the view.

However, we have notice that some other work, such as [36], reported the disadvantages of IC3 framework of the FlowDroid. Since the goal of our work is to detect user-unwared leakage among innumerous tainted flows but not to improve the ICC algorithm, we can still looking forward to the further improvement of the FlowDroid performance.

6.5 RQ5: Insight of Classified Flow Context

Table 4 presents the permission-categorized results of the metrics. Top 6 rows show source permissions, and the bottom 2 rows show the sink permissions. According to [15], we noticed that there are two interestring findings here. First, the general trend excluding some exceptions is that 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 the "Calendar" the lowest. In the sink permission categories, the accuracy number in "Internet" category is higher than the on in "SMS". Second, flows that have different semantics presentations have lower accuracy than those 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.

6.6 RQ5: Comparison with Alternative Classification Approaches

In this subsection, we would like to justify why we make such a choice in designing the semantic correlation module of *FlowCog*. Specifically, we want to select a better semantic correlation algorithm for the correlation module.

Table 5 shows the comparison results of different algorithms. The F_1 scores of efficient algorithms, including Logistic Regression (LR), Decision Tree (DT), and Naïve Bayes, are all bad, i.e., below 0.85. Linear Support Vector Machine (SVM) and Gradient Boosting (GB) perform better with 0.864 and 0.878, respectively, but are still not satisfying. Therefore, we evaluated combinations of different algorithms in Rows 6–11 of Table 5. The combinations of different algorithms obtained accuracy rates ranging from 80.6% to 90.2%. The combinations of Gradient Boosting and SVM is what we used to correlate the semantics in the *BoWSiM*. Among all the combinations that we evaluated, the *TextCNN* achieves the best results (95.4% accuracy and 0.953 F_1 score). Note that we also attempt the other deep-learning neural networks such as TextRNN and TextCNN-RNN joint network. The accuracies of those neural networks are around 95.5%, but the RNN and CNN-RNN joint network are much slower than TextCNN when we are training. In Figure 10, we present the performance metric of our TextCNN. The model can fastly converged within 1000 epoches. Finally, we chose TextCNN as the algorithm of our correlation module.

TABLE 5: Performance of Different Correlation Module

Algorithm	Precision	Recall	Accuracy	F_1 Score
Logistic Regression (LR)	84.2%	84.3%	81.9%	0.842
Decision Tree (DT)	73.8%	84.3%	73.8%	0.787
Naive Bayes (NB)	84.3%	83.3%	81.4%	0.838
Support Vector Machine (SVM)	86.8%	86.1%	84.5%	0.864
Gradient Boosting (GB)	84.2%	91.7%	85.3%	0.878
LR + DT	82.0%	84.5%	84.5%	0.832
LR + NB	84.5%	81.1%	80.6%	0.828
DT + SVM	85.3%	88.9%	86.0%	0.871
GB + NB	84.5%	88.9%	84.5%	0.868
GB + SVM	90.1%	93.1%	90.2%	0.916
TextCNN	96.4%	94.1%	95.4%	0.953

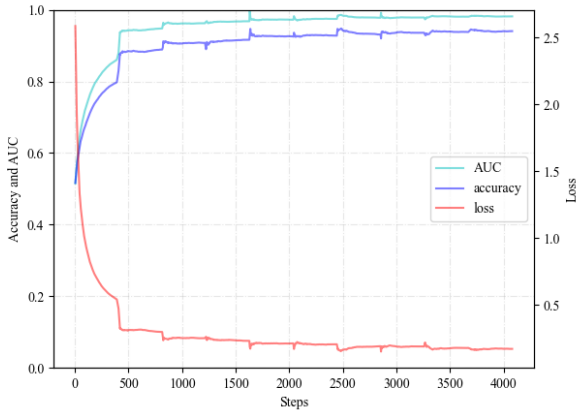
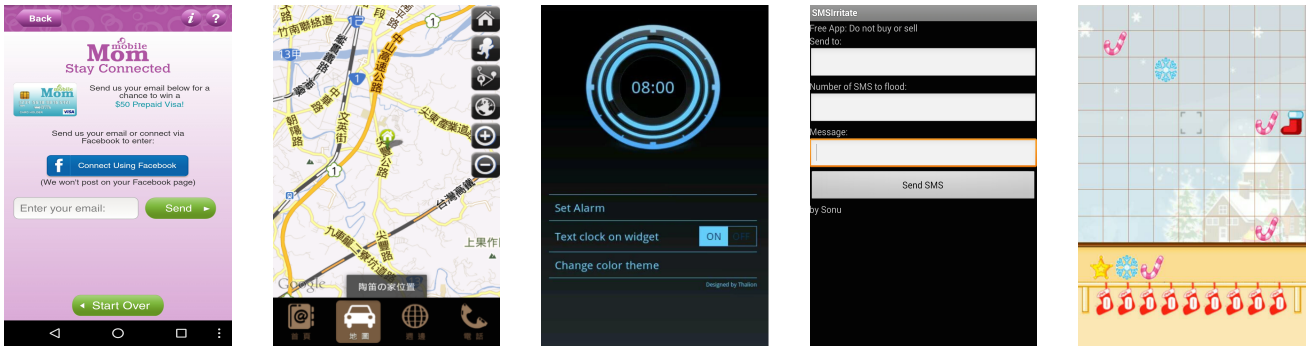


Fig. 10: *FlowCog* TextCNN Metrics of the Semantic Correlation

7 Case Study

In this section, we perform a case study on various 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 11a, 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



(a) Due Date Calculator (b) Home of Ocarina (c) Digital Clock Disc (d) SMS Irrirate (e) Merry Christmas

Fig. 11: Screenshots of Different Apps in the Case Study

from the database to the Internet. One flow is sending the user's email address to the Internet, and the other is sending URLs in another database to the Internet. *FlowCog* classifies the former as positive as flow contexts like "Send" and "Email Address" are available to the user, but the latter is 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 aims to deliver third-party ads. The app user has no knowledge of such an information leak. Note that existing app-level semantics correlation tools will not 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 11b, 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 a 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 11c 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. 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 11d, is a malicious app from the Drebin dataset [34], [35] with a positive flow leaking out user-specified information via a short message. This app aims 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 app's user 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 the Drebin dataset, which sends out users' information without their knowledge. Specifically, this app is a trojan that pretends to be a gaming app but hijacks the user's phone and leaks out confidential data when the user is playing the game. Figure 11e 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 the activity in the app and then extract semantics, which only includes 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?".

8 Limitations

In this section, we discuss the limitation in both dependency analysis and semantic correlation stages.

Dependency Analysis Stage. First we discuss the inter-component analysis performed in *FlowCog*. The static analysis of *FlowCog* relies on the FlowDroid [9]. Since currently FlowDroid has integrated the IC3, the FlowDroid is capable for the ICC analysis, which has the same ability of IC3 [19]. However, the inter-component analysis of IC3 is still limited by the ability of ICC method analysis [36], which is unable to resolve the undocumented ICC cases.

Second, 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* can 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.

Third, we discuss how clickjacking attacks, or in general UI redress attacks, influence our results. Simply put, these attacks are out of the 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.

Then, *FlowCog* requires to be able to detect any potential information leakage. Compared to the dynamic, static taint flow analysis provide a more comprehensive analysis of Android applications. However, *FlowCog* does not address the inherent shortcomings from static methodology, such as extracting runtime content. Thus, *FlowCog* does not reject optional dynamic analysis to detect information flows.

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 the semantics of Android apps are mostly provided in Java code.

Semantic Correlation Stage. To our knowledge, we also know other the state-of-the-art works in NLP, such as BERT [37], GPT-3 [38] reporting improvement in natural language semantic understanding. Theoretically, all those works can potentially help to improve the classification results. However, we do not have enough effort to try each of them. Thus we only selected simple models that can already reach the high accuracy.

9 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 [9], [10], [12]–[14], [19], [19], [36], [39]–[49]. FlowDroid [9] is a static precise taint analysis systems based on the 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. Recently, the FlowDroid has supported the inter-component analysis based on IC3 [19]. Static analysis systems Amandroid [10], DroidSafe [12], IccTA [11], and RAICC [36] are proposed to provide Android inter-component taint analysis to address this limitation. In addition to static analysis, dynamic analysis systems are also proposed to detect Android information flows. TaintDroid [14] conducts the taint analysis dynamically by proposing a customized Android framework. Uranine [45], on the other hand, detects information leakage by instrumenting the app without modifying the operating system. EdgeMiner [50] is an approach that detects implicit control flow transitions in the Android framework but does not analyze Android apps directly. Cadage [46] is a context-aware approach for GUI testing of Android application. It leverages probabilistic algorithm to select testing event to solve the non-determinism problem. Reardon et al. [48] summarize evidence of side and convert channels by analyzing unauthorized sensitive data sent to network through their dynamic analysis system. None of these works attempt to infer whether an Android app provides sufficient semantics for user to authorize information flows. That said, *FlowCog* can work with any such systems to determine whether enough semantics is provided.

Second, the Android app's execution context is an important indicator to analyze the app's behaviors. Several works are proposed to detect malicious Android apps based on execution contexts. SemaDroid [51] detects sensor contexts and reinforces sensor management policy-wise. 6thSense [52] performs observation for a specific set of sensorsáctivation activity-wise or task-wise. AppContex [53] 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 [54] identifies narrow conditional statements, called triggers, and infers possible suspicious actions based on these triggers. DroidSift [55] classifies Android malware using weighted contextual API dependency graphs. PikaDroid [49] identifies the malware based on contextual information from entrypoints and sensitive API. As a comparison, *FlowCog* goes beyond the 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 [6] 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 the app's descriptions and API documents, and then determines whether the descriptions justify the usage of certain permissions. Another research work, AutoCog [7], tried to solve a similar problem with NLP on descriptions but used a learning-based approach using the Android app's descriptions but not API documents. CHABADA [5] 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. [8] propose another NLP system that extracts the semantics from the 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 [56]. AsDroid [57] correlates the stealthy behaviors of Android apps, such as a malware, with the app's descriptions. DescribeMe [58] generates security-centric descriptions for Android Apps. As a comparison, *FlowCog* is the first system that analyzes the correlation between information flows and the semantics—*FlowCog* faces additional challenges such as extracting flow-specific semantics.

10 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* available at https://github.com/xcdu/FlowCog. Our evaluation results show that *FlowCog* can achieve a accuracy of 95.4% and an F_1 score of 0.953.

REFERENCES

[1] J. Y. Tsai, S. Egelman, L. Cranor, and A. Acquisti, "The effect of online privacy information on purchasing behavior: An experimental study," *Information systems research*, vol. 22, no. 2, pp. 254–268, 2011.

[2] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 3–14.

[3] L. Tsai, P. Wijesekera, J. Reardon, I. Reyes, S. Egelman, D. Wagner, N. Good, and J.-W. Chen, "Turtle guard: Helping android users apply contextual privacy preferences," in *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, 2017, pp. 145–162.

[4] I. Reyes, P. Wijesekera, J. Reardon, A. Elazari Bar On, A. Razaghpanah, N. Vallina-Rodriguez, S. Egelman *et al.*, ""won't somebody think of the children?" examining coppa compliance at scale," in *The 18th Privacy Enhancing Technologies Symposium (PETS 2018)*, 2018.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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," in *Acm Sigplan Notices*, vol. 49, no. 6. ACM, 2014, pp. 259–269.

[10] 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.

[11] 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.

[12] 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*, vol. 15, no. 201, 2015, p. 110.

[13] 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.

[14] 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.

[15] X. Pan, Y. Cao, X. Du, B. He, G. Fang, R. Shao, and Y. Chen, "Flowcog: context-aware semantics extraction and analysis of information flow leaks in android apps," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 1669–1685.

[16] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit," in *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, 2014, pp. 55–60.

[17] R. Mithe, S. Indalkar, and N. Divekar, "Optical character recognition," *International journal of recent technology and engineering (IJRTE)*, vol. 2, no. 1, pp. 72–75, 2013.

[18] Google Inc., "Intent | android developers," 2021, [Online; accessed 29-Nov-2021]. [Online]. Available: https://developer.android.com/reference/android/content/Intent

[19] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to android inter-component communication analysis," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 77–88.

[20] 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.

[21] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[22] S. Reese, G. Boleda, M. Cuadros, L. Padró, and G. Rigau, "Wikicorpus: A word-sense disambiguated multilingual wikipedia corpus," in *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*, 2010.

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

[24] 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.

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

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

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

[28] Gensim: topic modeling for human. https://radimrehurek.com/gensim/index.html.

[29] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

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

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

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

[33] fgwei, "Icc-bench," 2021, [Online; accessed 29-Nov-2021]. [Online]. Available: https://github.com/fgwei/ICC-Bench

[34] 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.

[35] 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)*.

[36] J. Samhi, A. Bartel, T. F. Bissyandé, and J. Klein, "Raicc: Revealing atypical inter-component communication in android apps," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1398–1409.

[37] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[38] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal,

A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[39] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, 2011, pp. 239–252.

[40] D. Octeau, S. Jha, and P. McDaniel, "Retargeting android applications to java bytecode," in *Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering*, 2012, pp. 1–11.

[41] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android: An essential step towards holistic security analysis," in *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, 2013, pp. 543–558.

[42] J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center, "Scandal: Static analyzer for detecting privacy leaks in android applications," *MoST*, vol. 12, no. 110, p. 1, 2012.

[43] M. Zhang and H. Yin, "Efficient, context-aware privacy leakage confinement for android applications without firmware modding," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014, pp. 259–270.

[44] A. Merlo and G. C. Georgiu, "Riskindroid: Machine learning-based risk analysis on android," in *Ifip international conference on ict systems security and privacy protection*. Springer, 2017, pp. 538–552.

[45] 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.

[46] H. Zhu, X. Ye, X. Zhang, and K. Shen, "A context-aware approach for dynamic gui testing of android applications," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, vol. 2. IEEE, 2015, pp. 248–253.

[47] J. Schütte, R. Fedler, and D. Titze, "Condroid: Targeted dynamic analysis of android applications," in *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*. IEEE, 2015, pp. 571–578.

[48] J. Reardon, Á. Feal, P. Wijesekera, A. E. B. On, N. Vallina-Rodriguez, and S. Egelman, "50 ways to leak your data: An exploration of apps' circumvention of the android permissions system," in *28th USENIX security symposium (USENIX security 19)*, 2019, pp. 603–620.

[49] J. Allen, M. Landen, S. Chaba, Y. Ji, S. P. H. Chung, and W. Lee, "Improving accuracy of android malware detection with lightweight contextual awareness," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 210–221.

[50] 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.

[51] Z. Xu and S. Zhu, "Semadroid: A privacy-aware sensor management framework for smartphones," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, 2015, pp. 61–72.

[52] A. K. Sikder, H. Aksu, and A. S. Uluagac, "{6thSense}: A context-aware sensor-based attack detector for smart devices," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 397–414.

[53] 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.

[54] 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.

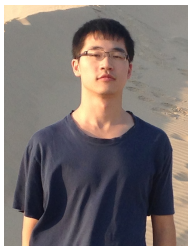
[55] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 1105–1116.

[56] 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.

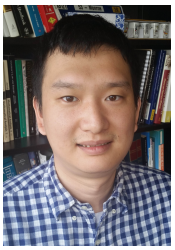
[57] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp.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

[58] 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.

Xuechao Du received his B.Eng. on computer science from Xian Jiaotong University, Xian, China, in 2016. He is currently a Ph.D. candidate in the college of Computer Science, Zhejiang University, China. His research interests include mobile security, Internet of Things security and software security based on program analysis.



Xiang Pan is a software engineer at Google. He earned his Ph.D. in computer science from Northwestern University. His research interests lie in cybersecurity with the special focus on: Web privacy/security, Android app privacy/security.

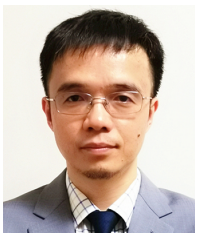


Dr. Yinzhi Cao is an assistant professor at Johns Hopkins University. He earned his Ph.D. in Computer Science at Northwestern University and worked at Columbia University as a postdoc. Before that, he obtained his B.E. degree in Electronics Engineering at Tsinghua University in China. His research mainly focuses on the security and privacy of the Web, smartphones, and machine learning.

Boyuan He is a postdoctoral research associate at Department of Electrical Engineering and Computer Science, Northwestern University. He earned his a Ph.D. in computer science from Zhejiang University. His research interests lie in cybersecurity with the special focus on: logic vulnerability detection, Android app security, blockchain security, IoT device security, malware detection and forensic analysis.

Yan Chen received his Ph.D. in Computer Science from University of California at Berkeley in 2003 and after that he joined Northwestern University USA where he became a Full Professor in 2014. His research interests are in security and measurement for networking systems. Based on Google Scholar, his papers have been cited over 14,000 times, and the h-index of his publications is 56. He is a Fellow of IEEE.

Gan Fang is a Sr. Staff Research Engineer at Palo Alto Networks Inc. He earned his M.S. in computer science from Northwestern University in 2017. He mainly works in areas including network traffic analysis & identification, payload inspection, protocol evasion prevention and SCADA protocols.



DaiGang Xu is a chief engineer and chief scientist of microservice platform of ZTE Corporation. He earned a master's degree in computer science from Sichuan University. His research interests lie in telecom software architecture with the special focus on: cloud native technology, microservice system, service mesh, telecom network cloudizatino and servitization, SDN/NFV/5G intelligent operation technology.

APPENDIX

```
1  class LoginActivity extends Activity {
2    void onCreate (Bundle state) {
3      // Source
4      String s = getLine1Number();
5      View et_username = findViewById(...);
6      Button bt_regist_submit = (Button)
       findViewById(...);
7      View et_regist_phone = findViewById(...);
8      et_regist_phone.setText(s);
9      et_username.addTextChangeListener(new
       RegistrationTextChangedWatcher() {
10         void afterTextChanged() {
11           // Guarding condition
12           if(et_username.getText().length() == 0)
13             bt_regist_submit.setEnable(false);
14           else
15             bt_regist_submit.setEnable(true);
16     }});
17     bt_regist_submit.setOnClickListener(new
       OnClickListener() {
18       // Activation Event
19       void onClick(View v) {
20         String phoneNumber = et_regist_phone.
       getText();
21         Intent i = new Intent(LoginActivity.this,
        RegActivity.class);
22         i.putExtra("phone_number", phoneNumber);
23         LoginActivity.this.startActivity(i);
24 }});}}
25 class RegActivity extends Activity {
26   void onStart() {
27     Intent i = getIntent();
28     String phoneNumber = i.getBoolExtra("
       phone_number");
29     // Sink is HttpClient.execute() in postData
30     S3ServerApi.postData(phoneNumber);
31 } }
```

Listing 1: Inter-Component Communication Example's Source Modified from S3 World Phone App.