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

Xuechao Du, Xiang Pan, Yinzi 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 F1 score of 0.953.

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

1 INTRODUCTION

A

NDROID 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 flows1 [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.

1. We use the following two terminologies, “information flow” and “data flow”, interchangeably in this paper.
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 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 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 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 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 its server after the user sees a registration page shown in Figure 1 and presses the “Submit” button.

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.getLineNumber(), 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 et1 (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
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 $e2$ and finds $et\_password$, 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.

3.1 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 $p_s \rightarrow p_e$ in the call graph of the target app where $p_e$ is a statement in the flow’s call path $(p_s \rightarrow p_k \rightarrow \ldots \rightarrow p_{sink})$. 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.

![Fig. 2: FlowCog Architecture](image)

![Fig. 3: Simplified Code Blocks of S3 World Phone App](image)

![Fig. 4: Call Path for the Data Flow in Figure 3](image)
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 $n_{source} \cdots n_k \cdots n_{sink}$, for any $n_k$, we define a conditional statement $c_k$—at least one branch of which does not contain $n_k$—as a guarding condition if either of the following is satisfied:

1. $c_k$ and $n_k$ are in the same basic block, or connected in the interprocedural Control Flow Graph (iCFG);
2. $c_k$ controls the activation events of the data flow via view dependency, i.e., $c_k$ 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
Interprocedural 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
12: method = getInvokedMethod(prevStmt) then
13: findGCHelper(curStmt, method.getFirstStmt(), iCfg, rs)
14: else
15: findGCHelper(curStmt, null, iCfg, rs)
16: end if
17: end if
18: end if
19: end for
20: return rs

Algorithm 3 The Algorithm of Finding Guarding Condition Helper Method

Input:
Target Statement: target
End Statement: endStmt
Interprocedural 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 if
15: end if
16: for parent in iCfg.getPreddecessors(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 setText(...). 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.

The Figure 5 presents 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 activation 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 referred from [11]. After Step 6, the ICC links will expand the original control 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

---

2. We present the source code in List 1 in Appendix.
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/this/startActivity() to onStart(), which is explicitly declared in RegActivity. Thus, the control flow graph connects the two activities LoginActivity and RegActivity. By locating the activation 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 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 $S3Server.Api.postData$ 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 BoWsSim. 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 BoWsSim, 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.
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 Dreibin dataset [34], [35]. Because we only parsed the English characters for evaluation from this app set, we call it the Monolingual App Set.

The latter large bilingual Android app set not only includes the English Android apps from Google Play App market and Dreibin 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 Dreibin) 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 F1 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%.

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

<table>
<thead>
<tr>
<th>No.</th>
<th>Correlation Model</th>
<th>App Diversity</th>
<th>Total Pos.</th>
<th>Total Neg.</th>
<th>TP</th>
<th>TN</th>
<th>FP</th>
<th>FN</th>
<th>Precision</th>
<th>Recall</th>
<th>Accuracy</th>
<th>F₁ Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BoWSiM</td>
<td>Monolingual(EN)</td>
<td>713</td>
<td>529</td>
<td>664</td>
<td>456</td>
<td>73</td>
<td>49</td>
<td>90.1%</td>
<td>93.1%</td>
<td>90.2%</td>
<td>0.916</td>
</tr>
<tr>
<td>2</td>
<td>TextCNN</td>
<td>Monolingual(EN)</td>
<td>713</td>
<td>529</td>
<td>670</td>
<td>511</td>
<td>18</td>
<td>43</td>
<td>97.4%</td>
<td>94.0%</td>
<td>95.1%</td>
<td>0.956</td>
</tr>
<tr>
<td>3</td>
<td>BoWSiM</td>
<td>Bilingual(EN+ZH)</td>
<td>5200</td>
<td>5215</td>
<td>3,883</td>
<td>3,538</td>
<td>1,677</td>
<td>1,347</td>
<td>69.8%</td>
<td>74.2%</td>
<td>71.0%</td>
<td>0.720</td>
</tr>
<tr>
<td>4</td>
<td>TextCNN</td>
<td>Bilingual(EN+ZH)</td>
<td>5136</td>
<td>5309</td>
<td>4,833</td>
<td>5,130</td>
<td>179</td>
<td>303</td>
<td>96.4%</td>
<td>94.1%</td>
<td>95.4%</td>
<td>0.953</td>
</tr>
</tbody>
</table>

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

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,
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 innumerable 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 interesting 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” is 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 F1 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 F1 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 epochs. Finally, we chose TextCNN as the algorithm of our correlation module.

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...
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 “irrate” the recipient. Although this is a malicious app, the flow is positive because the app’s user will understand that the recipient. 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], Ic- cTA [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 sensors’ activation 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. PIKA DROID [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
Our evaluation results show that FlowCog can achieve a score of 95.4% and an F1 score of 0.953.

References


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 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 cloudization and servitization, SDN/NFV/5G intelligent operation technology.
**APPENDIX**

```java
class LoginActivity extends Activity {
    void onCreate(Bundle state) {
        // Source
        String s = getLine1Number();
        View et_username = findViewById(...);
        Button bt_regist_submit = (Button) findViewById(...);
        View et_regist_phone = findViewById(...);
        et_username.setText(s);
        bt_regist_submit.setOnClickListener(new OnClickListener() {
            // Activation Event
            void onClick(View v) {
                String phoneNumber = et_regist_phone.getText();
                Intent i = new Intent(LoginActivity.this, RegActivity.class);
                i.putExtra("phone_number", phoneNumber);
                LoginActivity.this.startActivity(i);
            }
        });
        // Guarding condition
        if (et_username.getText().length() == 0) {
            bt_regist_submit.setEnable(false);
        } else {
            bt_regist_submit.setEnable(true);
        }
    }
}

class RegActivity extends Activity {
    void onStart() {
        Intent i = getIntent();
        String phoneNumber = i.getStringExtra("phone_number");
        // Sink is HttpClient.execute() in postData
        S3ServerApi.postData(phoneNumber);
    }
}

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