Profiling User-Trigger Dependence for Android Malware Detection☆

Karim O. Elish[a], Xiaokui Shu[a], Danfeng (Daphne) Yao[a,*], Barbara G. Ryder[a,], Xuxian Jiang[b,]

[a]*Department of Computer Science, Virginia Tech, 2202 Kraft Dr, Blacksburg, VA 24060, United States*
[b]*Department of Computer Science, North Carolina State University, Raleigh, NC 27606, United States*

Abstract

As mobile computing becomes an integral part of the modern user experience, malicious applications have infiltrated open marketplaces for mobile platforms. Malware apps stealthily launch operations to retrieve sensitive user or device data or abuse system resources. We describe a highly accurate classification approach for detecting malicious Android apps. Our method statically extracts a data-flow feature on how user inputs trigger sensitive API invocations, a property referred to as the *user-trigger dependence*. Our evaluation with 1,433 malware apps and 2,684 free popular apps gives a classification accuracy (2.1% false negative rate and 2.0% false positive rate) that is better than, or at least competitive against, the state-of-the-art. Our method also discovers new malicious apps in the Google Play market that cannot be detected by virus scanning tools. Our thesis in this mobile app classification work is to advocate the approach of *benign property enforcement*, i.e., extracting unique behavioral properties from benign programs and designing corresponding classification policies.

Keywords:
Malware detection, User-intention, Static program analysis, Android malware, User-trigger dependence

☆A preliminary version of the work appeared in the Proceedings of the IEEE Mobile Security Technologies (MoST) workshop, in conjunction with the IEEE Symposium on Security and Privacy. San Francisco, CA, USA. May 2012 Elish et al. (2012). This work has been supported in part by Security and Software Engineering Research Center (S^2ERC), an NSF sponsored multi-university Industry/University Cooperative Research Center (I/UCRC), NSF grant CAREER CNS-0953638, and ONR grant N00014-13-1-0016.

*Corresponding author. Department of Computer Science, Virginia Tech, 2202 Kraft Dr, Blacksburg, VA 24060, United States. Tel.: +1(540)231-7787

Email addresses: kelish@vt.edu (Karim O. Elish), subx@vt.edu (Xiaokui Shu), danfeng@vt.edu (Danfeng (Daphne) Yao), ryder@cs.vt.edu (Barbara G. Ryder), jiang@cs.ncsu.edu (Xuxian Jiang)

1. Introduction

Malicious mobile apps and vulnerable mobile computing platforms threaten the confidentiality of personal and organization data and device integrity Davi et al. (2010); Enck et al. (2010). Malicious applications can exfiltrate sensitive data, abuse of system resources, and disrupt the normal usage of the device. With the increased connectivity to organizational networks, vulnerable smartphones increase the attack surface of organizations, threatening the security of systems and data at a grand scale. Recent studies show that there exist hundreds of thousands of unique Android malware samples belonging to over 300 malware families forti-guard. Because of the pervasive use of Android as a mobile operating system (over 50% market share in western and some Asian countries), solutions for detecting malicious applications in the Android marketplace are urgently needed. Our work presents a new quantitative program analysis approach for detecting malicious Android applications that achieves a higher accuracy than previously reported classification methods.

Classification solutions have been proposed to model and approximate the behaviors of Android apps and distinguish malicious apps from benign ones. Classification decisions are made by analyzing apps' static (e.g., Grace et al. (2012b)) or dynamic (e.g., Amos et al. (2013)) behavior features. Static features can be extracted from intermediate code representations obtained through decompiling Android Dalvik bytecode. Dynamic features are collected by observing the run-time behaviors of the program. Various types of features can be extracted from Android permission, code, or execution for app classification.

The detection accuracy of a classification method depends on the quality of the features, e.g., how specific the features are. The accuracy of existing Android classification solutions is still far from ideal. The state-of-the-art classification with pure static features gives a false negative rate (i.e., missed detection, FN) of 9% Grace et al. (2012b). These features are extracted through data- and control-flow analyses. Hybrid features (i.e., a combination of static and dynamic features) extracted from programs give a better FN rate 4.2% Zhou et al. (2012) (e.g., dynamic features related to dynamic code loading and native code invocation). Most of the dynamic classification solutions give 10% or higher false positive rates (FP) while trying to maintain a reasonable FN rate, e.g., 10% FP in Shabtai et al. (2012) and 15% FP in Amos et al. (2013). The false positive rate tells the percentage of benign apps wrongfully classified as malicious.

This work presents a high-precision Android app classification method based on one complex feature that leverages the dependence effects of program behaviors. Specifically, we extract the definition-and-use (i.e., def-use) data dependence properties related to sensitive operations and their *user triggers* in the app. Smartphone apps (Android, iOS, or Windows Phone) are unique in their user-centered and interaction-intensive design, in which operations typically require initiation by users' specific actions (or triggers). Our classification leverages the dependence relations between user inputs/actions and sensitive API calls providing critical system functions. Our feature extracted from programs reflects the expected causal relations in the execution.

Our classification recognizes legitimate and desirable behavioral patterns in programs, as opposed to identifying malicious patterns. Those behaviors are commonly found in trustworthy programs, but not in malware. Our classification is based on whether or not a program possesses these benign properties.

Specifically, we analyze the def-use graph to extract a *TriggerMetric* feature for each API call. The TriggerMetric feature statically approximates whether or not the occurrences of the call (i.e., call sites) are triggered by the user. Specifically, the TriggerMetric value represents the number of *valid* call sites among all the call sites of a specific API. The validity of an API call is defined based on def-use semantics; a call is *valid* if at least one of the call's arguments depends on some user input(s). In other words, the TriggerMetric values of an app reflect the degree of sensitive operations that are triggered or intended by the user. The classification decision is made based on TriggerMetric values (i.e., an app is classified as malware if it has an overwhelming number of triggerless sensitive operations).

Our contributions are summarized as follows.

- We present a new Android app classification method that uses **one** complex feature rather than multi-feature as in the existing malware detection methods which focus on the presence of simple features such as permission or API call. The *TriggerMetric* feature captures the static dependence relations between user inputs/actions and sensitive operations providing critical system functions in programs. This feature is extracted through nontrivial Android-specific static program analysis and is used in several quantitative analytical methods.

- Our experimental evaluations on 2,684 free popular apps and 1,433 malicious apps suggest that our rule-based classification with the single feature of user-trigger dependence is very effective. It detects 97.9% of the malware apps with a low (2.0%) false positive rate.

- Our analysis reveals hundreds of malicious apps in the Google Play market, some of which were previously unreported and were not detected by any of the 48 *VirusTotal* [1] scanners.

The purpose of our work is *not* to advocate the use of fewer features in program classification. Multiple classification tools and features should be utilized to paint a comprehensive picture about a program.

Rather, our thesis in this mobile app classification work is to advocate the approach of benign property enforcement. Our analysis verifies whether or not a program is in compliance with our benign-property standards. In the face of rapid malware evolution, this type of benign-property enforcement may yield a more proactive defense than the malware-oriented detection approaches.

[1] https://www.virustotal.com/

2. Overview and Definitions

Our classification methodology aims at exposing possible privileged actions of apps that are not intended by the user and lack proper dependences in the code. In this section, we give the description of how the *trigger-based dependence* feature is extracted from programs through static program analysis. We also discuss several metrics formed from our feature analysis.

2.1. Data Dependence Graph

A data dependence graph (DDG) is a common program analysis structure which represents inter-procedural flows of data through a program Horwitz et al. (1990). The DDG is a directed graph representing data dependence between program instructions, where a node represents a program instruction (e.g. assignment statement), and an edge represents the data dependence between two nodes. The data dependence edges are identified by data-flow analysis. A direct edge from node n_1 to node n_2, which is denoted by $n_1 \rightarrow n_2$, means that n_2 uses the value of variable x which is defined by n_1.

Formally, let I be the set of instructions in a program P. The data dependence graph G for program P is denoted by $G = [I, E]$, where E represents the directed edges in G, and a directed edge $I_i \rightarrow I_j \in E$ if there is a def-use path from instructions I_i to I_j with respect to a variable x in P.

We show two DDG examples to motivate our data-flow analysis based on the dependence relations. The first example is a legitimate app for sending SMS messages. Figure 1 shows its partial def-use dependence graph. The graph indicates that the API call `sendTextMessage()` depends on the some inputs from the user, as one of its argument is entered by the user via text fields, through `getText()` API. There are direct dependence paths between user inputs (e.g., data and actions) and the `sendTextMessage()` API.

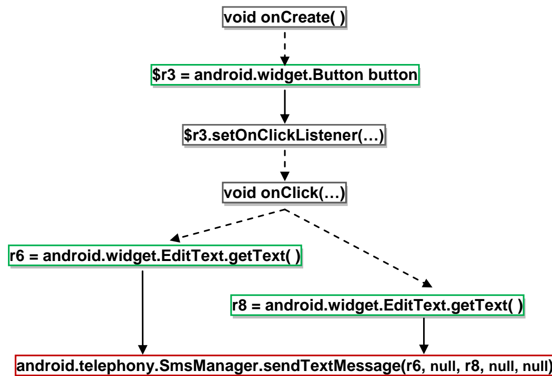

Figure 1: Partial abstract dependence graph for a legitimate app. `sendTextMessage()` has the required user dependence property. User triggers are shown in green nodes.

Another example is about a real-world Android malware `HippoSMS`, which affects Android smartphones by subscribing to premium SMS services. The

malware sends SMS messages to a hard-coded premium-rated number without the user's knowledge. Figure 2 shows a partial def-use dependence graph for HippoSMS. It shows the dependence relations associated with the arguments to a sensitive API call `sendTextMessage()`. Specifically, Figure 2 shows that `sendSMS(p0, p1, p2)` method is called with a hard-coded premium-rated number `1066156686` as its `p0` argument. The subsequent `sendSMS` method calls a sensitive API `sendTextMessage()` with the same hard-coded value `p0` as its `phoneNum` argument. There is no direct dependence path between the `sendTextMessage()` API call and any user inputs (e.g., data and actions).

We accurately extract these types of dependence properties and quantify them for classification. Existing program analysis solutions cannot be directly applied to solve the problem, in part because of the lack of proper handling of Android-specific features such as Intents. In our work, we formalize the security problem of dependence-based app classification, and design efficient algorithms for parsing large specialized data-dependence graphs for extracting the trigger-based dependence feature. We refine our data-dependence graph with reachability analysis obtained from control-flow analysis. The reachability analysis prunes unused code for high program analysis accuracy. The workflow of our analysis is shown in Figure 3.

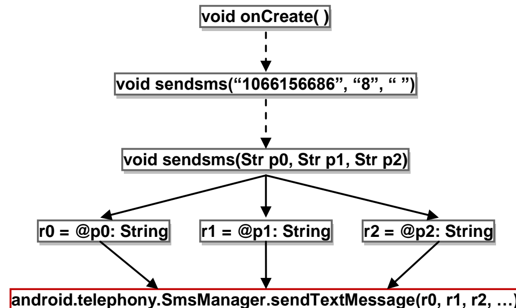

Figure 2: Partial abstract dependence graph for HippoSMS malware. There is no direct path showing a dependency between user triggers and `sendTextMessage()`.

2.2. TriggerMetric Tuple Per Operation

In this section, we give the definitions for the terminology used in our classification, including *operation*, *trigger*, *dependence path*, and *valid call site*. For each operation in a program, we give our definition for the TriggerMetric tuple, which represents properties associated with call sites of the operation.

An ***operation*** is an ***API call*** which refers to a function call providing system service such as network I/O, file I/O, telephony services in the program. We focus on a subset of function calls – the critical API calls that can be used for accessing private data and utilizing system resources.

Examples of the operations in our analysis are send/receive network traffic, create/read/write/delete operations for files, insert/update/delete operations

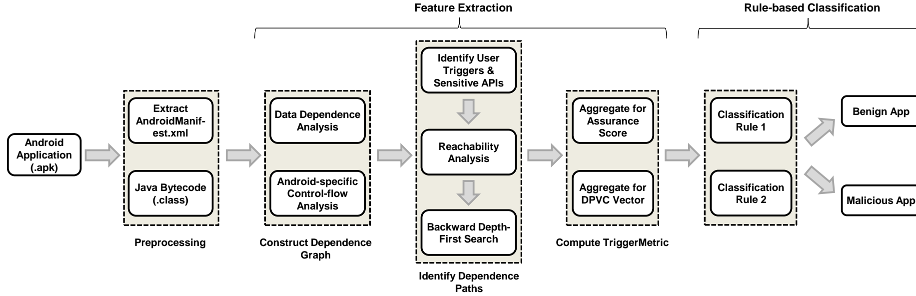

Figure 3: Workflow of our analysis.

in database and content provider, execute system commands using `java.lang.Runtime.exec`, access and return private information such as location information and phone identifiers, and send text messages in telephony services.

A ***trigger*** refers to a user's input or action/event on the app. A trigger is a variable defined in the program. For example, the user's input may be text entered via a text field, while the user's action/event is any click on UI element, such as a button. Relevant API calls in UI objects that return a user's input value or listen to user's action/event are defined as triggers.

Our classification is based on analyzing unauthorized privileged operations that are not intended by the user. Because the analysis is automated (i.e., without any user participation), user-intention needs to be approximated. In our analysis user-intention is embodied in the trigger variables. We specify the names of functions corresponding to triggers and operations in the program analysis.

A ***valid dependence path*** is a (directed) dependence path between a trigger and an operation in a data dependence graph (DDG). In our static data-flow semantics, the path specifies a definition-and-consumption (*def-use*) relation, where a trigger is defined and later used as an argument to an operation. The existence of a valid dependence path means that the operation depends on a user trigger.

Figure 4 illustrates two different operations c and c' in a program, each having two call sites (i.e., each call occurs twice in the program), s_1 and s_2 for c, s'_1 and s'_2 for c'. Three dependence paths are valid, with proper user triggers on the paths, whereas a valid dependence path for call site s'_2 does not exist.

The trigger may be transformed before being used as an argument in the operation, thus the dependence path between them may be long. In Section 3 we present our detailed program analysis and graph algorithms.

A ***valid call site*** s of an operation c is a call site that has a valid user-trigger dependence path. A *call site* is the occurrence of an operation. An operation may have one or more call sites in a program.

Definition 1. TriggerMetric feature *is a two-item tuple* $< k, l >$ *for an operation* c *in a program, where*

- *k is the number of valid call sites of operation c, and*
- *l is the total number of call sites of operation c.*

For the example in Figure 4, the TriggerMetric values for operations c and c' are $< 2, 2 >$ and $< 1, 2 >$, respectively. For an app with n distinct operations, there are n TriggerMetric tuples associate with it, $< k_1, l_1 >$, ..., $< k_n, l_n >$, one corresponding to each operation.

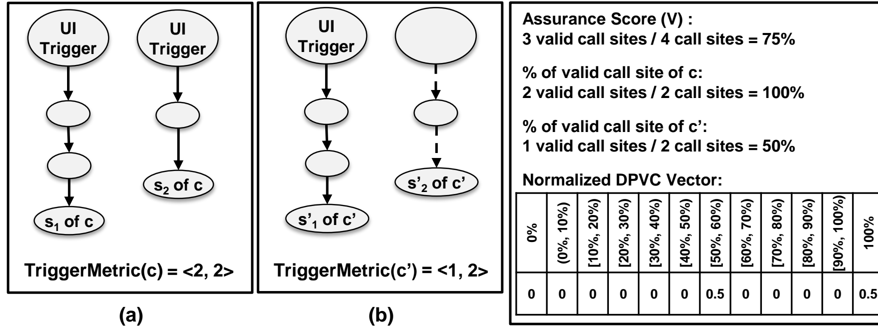

Figure 4: Illustration of dependence paths and various metrics for a program having two distinct operations c and c'. Each operation has two call sites s_1 and s_2 and s'_1 and s'_2, respectively. A solid line represents the existence of a dependence path from some user trigger to a call site. A dashed line represents that none of the call site's dependence paths has a user trigger.

2.3. Aggregated Metrics

One can compute several useful values aggregated from the n TriggerMetric tuples of a program. These aggregated metrics provide a behavioral summary of the program. Intuitively, the *assurance score* V is a single value for an app representing the portion of call sites that are intended by the user across all operations in the app.

Definition 2. Assurance score $V \in [0\%, 100\%]$ *of a program is the percentage of valid call sites out of the total number of call sites across all the operations. Given the n TriggerMetric tuples $\{< k_i, l_i >\}$ of a program, where k_i is the number of valid call sites and l_i is the number of total call sites for operation i, and n is the total number of distinct operations, V is computed as follows.*

$$V = \frac{\sum_{i=1}^{n} k_i}{\sum_{i=1}^{n} l_i} \tag{1}$$

For the example in Figure 4, $V = \frac{3}{4}$, as there are total 4 call sites in the program, among which 3 are valid.

One can also compute the distribution associated with TriggerMetric values in a program, which provides useful insights into the program's behaviors.

Definition 3. DPVC Vector W *of a program is the normalized* **D***istribution of the* **P***ercentages of* **V***alid* **C***all sites per operation. For operation* i*, the percentage of valid number of call sites is defined as* $\frac{k_i}{l_i}$*, where* k_i *is the number of valid call sites and* l_i *is the number of total call sites for the operation* i*. Let* n *be the total number of distinct operations in the program.*

Each percentage value determines the bin whose contents are augmented by one. After all percentage values are distributed, the value of each bin is divided by n*, the total number of operation in the program. This yields a normalized distribution. Specifically, the distribution of the* n *percentage values* $\{\frac{k_1}{l_1}, \ldots, \frac{k_n}{l_n}\}$ *is represented by the following 12 bins: 0%, (0%, 10%), [10%, 20%), [20%, 30%), ..., [90%, 100%), 100%.*

For the example in Figure 4 ($n = 2$), the percentages of valid number of call sites for the two operations (c and c') are 100% ($\frac{2}{2}$) and 50% ($\frac{1}{2}$), respectively. Thus, most of the corresponding DPVC vector is 0, except for bins [50%, 60%) and 100%, i.e., one count in the [50%, 60%) bin, and one count in the 100% bin. After normalization, the entry for both the 100% bin and [50%-60%) bin is 0.5. Therefore, the final normalized distribution vector is {0, 0, 0, 0, 0, 0, 0.5, 0, 0, 0, 0, 0.5}, whose components are summed to 1.

The DPVC vector is computed from the TriggerMetric feature. Intuitively, it provides the in-depth statistics on the dependence-based validity of the calls in the program. The vector is used in our classification in Section 4, where we compare the DPVC vector of an unknown app with ones of known malware apps to infer their behavior similarities.

2.4. Program Analysis for Feature Extraction

The TriggerMetric feature is extracted from programs through static program analysis. In this section, we justify our use of data-flow analysis (as opposed to control-flow analysis) for this purpose. Our method tracks how a user's input propagates throughout the program using *data-flow analysis*. Alternatively, one may attempt to capture how the user control action leads to a sensitive API call, which requires *control-flow analysis*.

For our trigger-based dependence analysis, data-flow analysis is more appropriate than control-flow. For example, control-flow analysis cannot be used to track the user's input (data) that is used as arguments in sensitive API calls. However, data-flow analysis alone may overestimate the dependences due to the lack of the control analysis on branches (e.g., `if`). In this work, our feature is extracted from data-flow dependence analysis, which is coupled with event-specific control-flow dependence analysis. Our approach can be generalized to comprehensive control-flow analysis for improved accuracy.

Our dependence analysis tracks the propagation of triggers through events, including Android Intent. Intent is an event-based mechanism for communication between applications or components (*Activity, Service, Receiver*) in Android. For example, information entered by the user in one *Activity* may be

passed through an Intent to another *Activity* or *Service* for processing. Therefore, the dependence graph needs to be augmented in order to obtain the complete set of operations that depend on trigger variables through events. Without this expansion, the dependence analysis may underestimate the dependence relations (i.e., fail to report legitimate trigger-operation dependence relations). Because of our focus is on dependences related to user activities, we perform Intent-specific control-flow analysis, as opposed to general control-flow analysis.

Next, we give a detailed description of the techniques used in our program analysis. The program analysis outputs TriggerMetric values for all the sensitive operations in the program. Then in Section 4, we present our classification method based on the TriggerMetric values. Our evaluation results are given in Section 5.

3. Feature Extraction Using Dependence Analysis

We present in detail our technique used for extracting the TriggerMetric feature from Android applications. To that end, we generate and analyze the data dependence graph, including *i)* the general data-flow dependences, *ii)* the event-specific data dependence analysis for handling Android Intent and gathering comprehensive data dependence information, *iii)* reachability analysis for pruning unused code, and *iv)* backward depth-first search for finding dependence paths and computing a TriggerMetric for each operation.

Our program analysis takes as inputs the trigger set and the operation set, which are manually selected based on their semantics. The output of the program analysis is a set of TriggerMetric values $\{< k_c, l_c >\}$, one value for each sensitive operation c, e.g., `sendTextMessage()`.

The pseudocode of our procedure for computing TriggerMetric values of a program is shown in Algorithm 1.

Algorithm 1 ComputeTriggerMetric

Input: $A \leftarrow$ {App code}
$U \leftarrow$ {Set of user triggers of an app}
$M \leftarrow$ {Set of operations call sites of an app}
entryPoints of the app
Output: TriggerMetric set $S = \{< k_i, l_i >\}$, where k_i is the number of valid call sites and l_i is total number of call sites for operation i

1:
2: $RUT \leftarrow \emptyset$ //a list of Reachable User Triggers (RUT)
3: $RCS \leftarrow \emptyset$ //a list of Reachable Call Sites (RCS)
4: $S \leftarrow \emptyset$ //a set of TriggerMetric values
5: parse AndroidManifest.xml file
6: $G \leftarrow$ ConstructDataDependenceGraph(A)
7: *(RUT, RCS)* $\leftarrow$ *identifyReachableComponents*(U, M, entryPoints)
8:
9: **for** each *operation* i **do**
10: $\quad C_i \leftarrow$ {set of call sites of operation i} $\in RCS$
11: $\quad k_i \leftarrow checkPathExistence(RUT, C_i, G)$
12: $\quad l_i \leftarrow |C_i|$
13: $\quad S \leftarrow S \cup < k_i, l_i >$
14: **end for**
15:
16: **return** TriggerMetric set S
17:
18: **procedure** *identifyReachableComponents*(U, M, entryPoints)
19: $\quad G' \leftarrow$ ConstructControlFlowGraph(A)
20: $\quad$ **for** each $u \in U$ **do**
21: $\quad\quad$ perform DepthFirstSearch(u, entryPoint, G')
22: $\quad\quad$ **if** a path $\in G'$ exists between u and entryPoint **then**
23: $\quad\quad\quad RUT \leftarrow RUT \cup \{u\}$ //Reachable User Triggers (RUT)
24: $\quad\quad$ **end if**
25: $\quad$ **end for**
26: $\quad$ **for** each $m \in M$ **do**
27: $\quad\quad$ perform DepthFirstSearch(m, entryPoint, G')
28: $\quad\quad$ **if** a path $\in G'$ exists between m and entryPoint **then**
29: $\quad\quad\quad RCS \leftarrow RCS \cup \{m\}$ //Reachable Call Sites (RCS)
30: $\quad\quad$ **end if**
31: $\quad$ **end for**
32: $\quad$ **return** RUT and RCS
33: **end procedure**
34:
35: **procedure** *checkPathExistence*(RUT, C_i, G)
36: $\quad k_i \leftarrow 0$ //initialize k_i for operation i
37: $\quad$ **for** each $c \in C_i$ **do** //for each call site of operation i
38: $\quad\quad$ **for** each $u \in RUT$ **do** //for each user trigger
39: $\quad\quad\quad$ perform backward DepthFirstSearch(c, u, G)
40: $\quad\quad\quad$ **if** a directed path $\in G$ exists between c and u **then**
41: $\quad\quad\quad\quad k_i + +$
42: $\quad\quad\quad\quad$ break
43: $\quad\quad\quad$ **end if**
44: $\quad\quad$ **end for**
45: $\quad$ **end for**
46: $\quad$ **return** k_i
47: **end procedure**

We first describe our construction of the dependence graph based on explicit def-use relations. The basic DDG graph is then augmented in order to capture def-use relations due to events.

3.1. General-Purpose Data-Flow Dependence

We use data-flow analysis to construct the data dependence graph (DDG) with intra- and inter-procedural call connectivity information to track the dependences between the definition and use of user-generated data in a given

program. The intra-procedural dependence edges are identified based on local use-def chains. On the other hand, the inter-procedural dependence edges are identified based on constructing a call-site context-sensitive call graph supported by *points-to analysis* to build accurate call graphs. *Context-sensitive* analysis differentiates calling contexts of a function during analysis. *Context-insensitive* analysis analyzes a function summarizing over all calling contexts.

Thus, a context-insensitive analysis may not provide as accurate a solution.

The above general-purpose data-flow analysis does not cover the data-flow associated with events, as Android event communications are usually implicit. To achieve a comprehensive dependence coverage, we describe our technique for the necessary event-specific dependence analysis next.

3.2. Augmentation with Event-Specific Data Dependence

Our augmented analysis handles two types of events – *i)* implicit method invocation (e.g., through listeners in GUI) and *ii)* Android-specific Intent-based inter-app or inter-component events. Our approach is to perform necessary control-flow analysis, which finds *bridges* between disjoint graph components, so that one can obtain the complete reachability of triggers. We describe our Android Intent-based dependence analysis that tracks the control-flow among Intent-sending methods in intra- and inter-application communication. This Intent-specific control-flow analysis is necessary for capturing data dependence relations between triggers and operations across multiple apps and their components.

Android Intent can declare a component name, an action and optionally includes data or extra data. For example, an Intent can be used to start a new activity by invoking the `startActivity(Intent i)` or `startActivityForResult (Intent i, ...)` methods. An Intent should be sent to a target component by matching the Intent's fields with the declaration of the target component in the manifest. Android Intents can be used for explicit or implicit communication. An explicit Intent specifies that it should be delivered to a particular component specified by the Intent, whereas an implicit Intent requests the delivery to any component that supports a desired operation.

For *explicit Intent*, where the target component name is specified, we first identify the source component and the target component that are linked through an Intent object. This step pinpoints the Intent creation and sending methods (e.g., `startActivity(Intent i)` and `sendBroadcast(Intent i)`) to capture the control-flow dependences between the source and target components. In particular, we analyze the Intent object constructor to extract the name of the target component if it is provided. If it is not provided, we search the parameters in the `setClass()`, `setComponent()` or `setAction()` methods on the Intent object, which specify the target's name to obtain the target component. Given this information, the dependence graph is augmented by adding a directed edge from the Intent-sending method of the source component to the target component. This analysis is performed for all explicit Intents created in a given application.

For an *implicit Intent*, the target component can be any component that declares its ability to handle a specified action. The target component is determined by the Android system based on the manifest file. We handle the implicit Intent by analyzing the AndroidManifest.xml file to extract a list of components with their actions to identify the target component. Implicit method invocation, such as those in the GUI, must be accounted for in the dependence graph. Our approach is to connect the dependent calls to the relevant API calls related to threads and listeners with their callee in the graph. For example, `Button.setOnClickListener()` is linked with an implicit call to its event handler implementation `onClick()`. We identified a list of all event handlers from Android developer documentation for our analysis. These methods effectively augment the general-purpose data dependence graph with the necessary Android event-specific data-flow information.

Obfuscation, Java reflection, and dynamic code loading cannot be analyzed statically. Dynamic analysis approaches (e.g., Newsome and Song (2005); Yin et al. (2007)) are needed to extract related runtime behavioral features.

3.3. Reachability Analysis

The above operations produce a flow- and context-sensitive data-flow dependence graph with intra- and inter-procedural dependence analysis, and intra-and inter-application Intent-based dependence analysis. We then perform a reachability analysis for the app in order to remove unreachable code "dead code". Unreachable code is a portion of the program which contains classes/methods that are not executed. To that end, we construct an inter- and intra-procedural control-flow graph which shows all the possible execution paths. Given this control-flow graph and the list of user triggers and sensitive API calls, we perform reachability analysis to identify reachable user triggers and sensitive API calls from the entry points of the app. Specifically, we trace forward from the given entry point looking for the identified user triggers and sensitive API calls. For example, we perform reachability analysis to check whether a certain user trigger, e.g. click button, is reachable from the main activity. An activity is a visible portion of an application which handles user interaction.

There might be some user triggers inside other activities, but these activities never get executed or called from the main/parent activity. Hence, there is no reachable path from the entry point and these user triggers, and they can be safely ignored to increase the precision of our analysis. Similarly, some sensitive API calls may not be reachable from the entry points and never get executed. For example, a sensitive API `getLastKnownLocation()` in a tool app is unreachable from the apps entry points, and therefore will not be executed. Thus, we ignore and call it unreachable sensitive API call.

On the other hand, we call user trigger or sensitive API call reachable if there is a reachable path from the given entry point to this user trigger or sensitive API call. For example, assume that there is a sensitive API `sendTextMessage()` identified in a service component in app SendSMS. A service is an invisible portion of an application which performs background task. This service will be

called from the main activity upon user clicks on a button. In this case, the sensitive API identified inside the service component will be executed. Thus, there is a reachable path from the main activity entry point to this `sendTextMessage()`, and hence we call it reachable sensitive API call.

As explained above, some user triggers and sensitive API calls may not be reachable and hence can be ignored in our analysis. Our subsequent dependence analysis will only be performed on reachable components. The reachability analysis increases the analysis precision by excluding unreachable code.

3.4. Finding User-Trigger Dependence Paths

Once the dependence graph is constructed, the next step is to identify paths between user trigger and sensitive API call pairs. We scan the graph for the occurrences or call sites of sensitive operations. In Algorithm 1, `checkPathExistence()` performs this task by performing backward depth-first traversal. For *each call site* s_i of an operation c, we perform the backward tracing from s_i on the dependence graph searching for any user triggers on the dependence paths. For each c, we record the *valid* number k_c of call sites, and the *total* number l_c of call sites. $< k_c, l_c >$ is output as the TriggerMetric of the call c, according to Definition 1.

Our implementation of the static analysis framework utilizes libraries in Soot, a static analysis toolkit for Java soo. Our framework analyzes Java bytecode or source code.

Our DDG construction improves the def-use analyses provided by Soot [2]. Our prototype propagates def-use relations across the boundaries of methods. Our current prototype dose not analyze native libraries. Yet, our approach can be generalized to analyze native code.

4. Classification Method

The classification decisions are based on the assurance score V and DPVC vector W of an app. An app is classified as either *benign* or *malicious.* These values are computed from the extracted TriggerMetric tuples ($< k_i, l_i >$) of the app, according to Definitions 2 and 3. Because of the simplicity of our feature, our classification is based on rules. In addition to classification decisions, our analysis also reports the names of operations with invalid call sites in the program.

Specifically, given the TriggerMetric values obtained from the program analysis, our classification has three steps: *i)* computing V and W, *ii)* preliminary classification based on V with respect to a pre-defined threshold T, and *iii)* further classification based on the weighted similarity analysis between vector W and those of known malware samples. In the next section, we present our two classification rules.

[2] We augmented Soot libraries to support the inter-procedural call dependence analysis.

4.1. Our Classification Rules

Classification with assurance score. The threshold-based classification Rule 1 aims to detect apps that have low assurance scores, indicating the existence of a large portion of invalid call sites without proper user triggers.

Rule 1. *Given the assurance score V of an Android app and an assurance threshold $T \in (0, 100\%]$, if $V < T$, then the app is classified as malware. Otherwise, it is classified as benign.*

Clearly, the choice of T affects the accuracy of the classification. In our experiments in Section 5, we found that a threshold of 75% gives a proper balance between the false positives (FP) and false negatives (FN). Probable malware needs to be further inspected.

For each app, we also applied the similarity-based classification rule.

Weighted similarity analysis on DPVC vector. This classification compares the DPVC vector of an app with the DPVC vectors of known malware samples. The purpose is to detect the apps who have *similar distributions with malware* in terms of the valid call sites. To that end, we first computed the DPVC vector W^i for each malware $i \in [1, m]$ in a known malware sample set of size m. Then, we computed the average DPVC vector, which is denoted by M; that is, for each item M_j in vector M, M_j is computed as in Equation 2.

$$M_j = \frac{\sum_{i=1}^{m} W_j^i}{m} \tag{2}$$

Vector M represents the *average* distribution of the percentage of valid call sites per operation among the known malware.

Rule 2. *Given the DPVC vector W of an app, the average malware DPVC vector M, a similarity function f, and a threshold T', if $f(W, M) \geq T'$, then the app is classified as malware. Otherwise, it is classified as benign.*

Any similarity function may be used on DPVC vectors. In our experiments, we used a weighted cosine similarity function Tan et al. (2006). The function computes the cosine similarity between vectors W and M, while applying weights to the ranges with smaller percentage values, namely 0% and (0, 10%). The weights are computed based on an exponential function 2^x and then are normalized.

The reason for choosing the exponential weight function for this similarity measure is that we observed that the malware apps have a distinct distribution pattern from the legitimate apps towards the low percentage region, as shown in Figure 5. The weights amplify this distinction in the classification.

Definition 4. *A program is classified as benign if it is classified as benign by both Rule 1 and Rule 2. Otherwise, it is classified as malicious.*

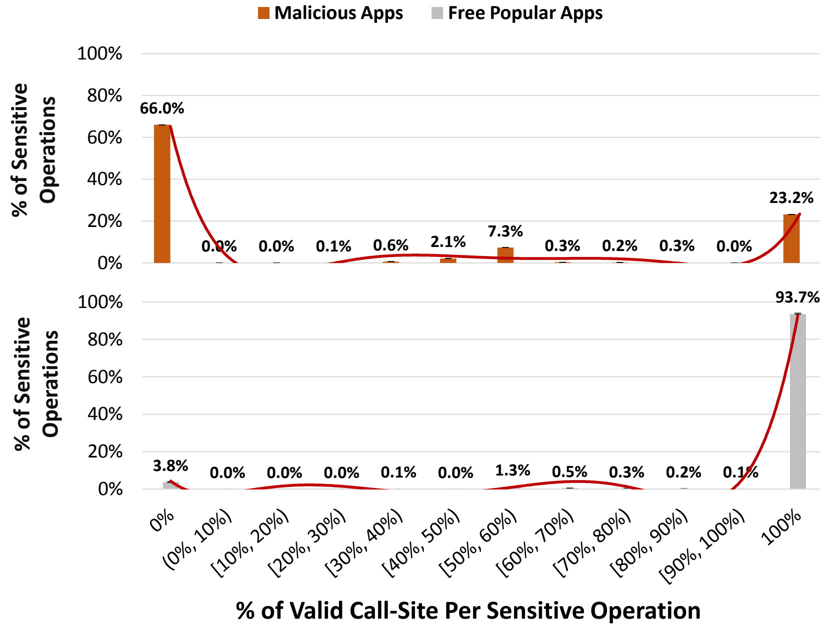

Figure 5: Averaged DPVC vectors representing a fine-grained distribution of per-operation valid call sites for 1,433 malware apps (top) and 2,684 free popular apps (bottom).

Our evaluation indicates the effectiveness of the above classification rules on the thousands of apps studied. We also painstakingly performed necessary manual inspections on some apps to validate our results and identified the causes of inaccuracies.

In the next section, we present category of features derived from our TriggerMetric value which can be used for classification as well.

4.2. Variations of Classification Rules

Our classification rules are based on aggregated statistics on valid call sites of a program. One can define other classification rules using the TriggerMetric values $\{< k, l >\}$ of a program. These rules may reflect different degrees of user-trigger dependence that is required in a trustworthy application.

To demonstrate the generality of the TriggerMetric feature, in this section we describe two examples of such classification rules, namely *All-Valid-Call-Sites Rule* and *Any-Valid-Call Site Rule*. Both rules defined below are based on the number of *valid* call sites k_i with respect to the *total* number of call sites l_i for an operation i in the program.

Rule 3. All-Valid-Call-Sites Rule. *A program is classified as benign, if and only if all the call sites of all the sensitive operations are valid, i.e., having user-trigger dependence. If* $k_i = l_i$ $\forall$ *sensitive operation* i*, then the program is benign. Otherwise, the program is classified as malicious.*

This above rule is equivalent to setting assurance threshold T to 100% in our classification Rule 1. In our experiments, there are 80.5% (2162) of apps that have 100% assurance scores. We conjecture that such a rule leads to low or zero missed detection, but many false positives.

A more relaxed classification rule can be defined below, which only requires *at least one* valid call site per sensitive operation.

Rule 4. Any-Valid-Call-Site Rule. *A program is classified as benign, if for each sensitive operation there is at least one valid call site. If* $k_i \geq 1 \; \forall$ *sensitive operation* i*, then the program is classified as benign. Otherwise, the program is classified as malicious.*

For the example in Figure 4, this program is classified as malicious by Rule 3 and benign by Rule 4. In-depth comparison of the impact of these various classification rules and thresholds on Android security will be our future work.

In our experimental evaluation, the classification decisions are based on Rule 1 and Rule 2.

5. Experimental Evaluation

The objective of our evaluation is to answer the following questions:

1. Do the distributions of the assurances scores of malware and benign apps significantly differ?
2. What is the false negative (i.e., missed detection) rate when classifying known malware samples?
3. Can our method discover new malware apps that have not been previously reported?
4. What are the reasons for false positives?

5.1. Experiment Setup

We performed an evaluation with 1,433 Android malware apps collected by Zhou and Jiang (2012) and VirusShare [3]. The known Android malware apps perform malicious functionalities, such as sending unauthorized SMS messages (e.g., *FakePlayer*), subscribing to premium-rate messaging services automatically (e.g., *RogueSPPush*), listening to SMS-based commands to record and upload the victim's current location (e.g., *GPSSMSSpy*), stealing users' credentials (e.g., *FakeNetflix*), and granting unauthorized root privilege to some apps (e.g., *Asroot* and *DroidDeluxe*) [4].

We also evaluated 2,684 free popular real-world Android apps from Google Play market, covering various application categories. These free apps include those with different levels of popularity as determined by the user rating scale.

[3] http://virusshare.com/

[4] The malware naming convention follows Zhou and Jiang (2012).

In particular, we used 1,039 high popularity apps, 713 intermediate popularity apps, and 932 low popularity apps. We assumed that the trustworthiness of these free apps is unknown and they may be malware or may contain malicious components. We converted Android app code (apk) from the .dex format to .class files using the *Dare* tool Octeau et al. (2012) and extracted features from the Java bytecode.

Averaged DPVC vector of known malware. We computed the DPVC vector for each of the 1,433 malware samples, and then computed their average DPVC vector according to Equation 2. The average malware DPVC vector approximates the distribution of valid call sites in malicious apps. It was used for the similarity test of unknown apps in Rule 2.

Thresholds for classification rules. For our two classification rules (Section 4), we choose the assurance threshold T to be 75% for Rule 1 and the similarity threshold T' to be 0.8 for Rule 2. Empirical results showed that these values provide a high detection rate without producing excessive false alerts.

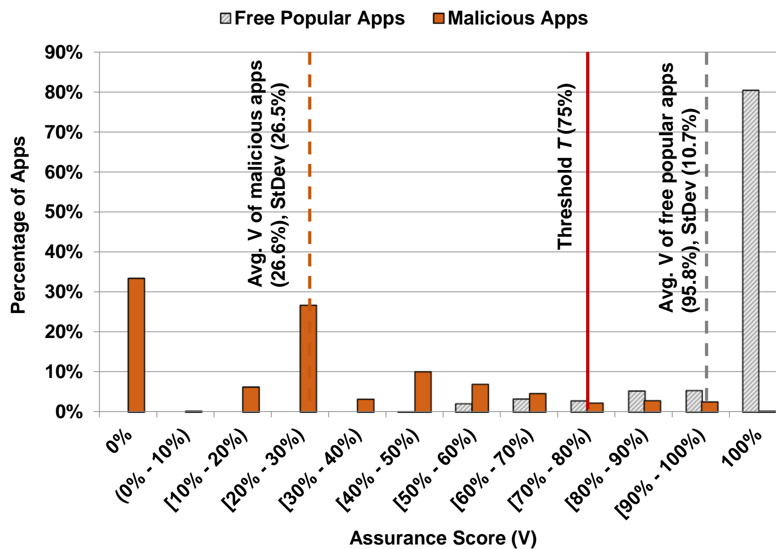

Figure 6: Distinct distributions of assurance scores (V) for known malicious apps and free popular apps.

5.2. Known Malicious Apps

Assurance Scores of Known Malware Most of the malware apps have low assurance scores, indicating that a significant number of sensitive API calls are made without proper user triggers. Invalid call sites that we observed include those for writing and sending information through the network, sending unauthorized SMS messages, executing system commands, and accessing user's private data. E.g., *Asroot* and *BaseBridge* use `Runtime.exec()` to execute system commands without valid user triggers.

We found that 479 malware apps out of 1,433 apps have 0% assurance scores. The rest of the 954 apps have positive assurance scores. Among them, many malware apps are repackaged from benign apps [5], e.g., *ADRD*, *DroidDream*, and *Geinimi*. Malware writers bundle malicious code with existing benign apps. Repackaging explains our observation that a significant number of malware apps (954 out of 1,433) have non-zero assurance scores. Positive assurance scores indicate that a portion of the sensitive operations in these malware apps exhibit the required dependences on user triggers.

`FakeNeflix` is the only malware app that has a 100% assurance score. `FakeNetflix` is a phishing app, which provides a fake user interface to trick the user to enter her or his Netflix credential. This type of phishing malware circumvents virtually all behavior-based detection approaches, including ours. App certification and user education are more effective defenses than program analysis for this type of social engineering malware.

The detailed distribution of the assurance scores for the known malicious apps can be found in Figure 6.

Classification Results on Known Malware

Rule 1 (V)		Rule 2 (DPVC)	
Malicious	Benign	Malicious	Benign (FN)
92.5%	7.5%	5.4% out of 7.5%	2.1% out of 7.5%

Table 1: Summary of classification results on 1,433 known malware apps. Rule 2 is applied to the apps that are classified as benign by Rule 1. The false negative (FN) rate refers to the portion of malware apps classified as benign by both rules and is 2.1%.

The classification results on known malware apps are given in Table 1. Using assurance scores, Rule 1 labels most (92.5%) of the samples as malicious, as they have lower-than-75% V values. Rule 1 labels 108 apps (7.5%) as probably benign. Using DPVC vectors, Rule 2 labels malicious for 5.4% (77) apps out of the 108 probably benign cases, as these apps have low percentages for valid call sites per operation. Thus, **we correctly detect 97.9% of the 1,433 malware samples.** The false negative rate is 2.1%, i.e., 31 malware apps are misclassified as benign.

The main reason for misclassification is malware repackaged from existing benign code, resulting in malware with profiles similar to benign apps. For example, one of the 31 undetected malware apps is `DroidKungFuSapp`, which contains malicious code bundled with `com.aijiaoyou.android.sipphone` (an app for learning Chinese). As a result, this malware app has a high assurance score V of 85.7% and a low similarity value (0.015) with known malware.

There are two possible countermeasures to combat the misclassification of

[5] The problem of detecting repackage apps (e.g., Crussell et al. (2012)) has a more specific goal from our general app classification. It typically requires graph-based pair-wise app similarity analysis.

repackaged malware apps. The first countermeasure is to adjust the rules thresholds used for the classification. For example, we set a threshold for rule 1 (assurance score V) to 75% in our evaluation. One can raise this threshold to be 90% or more. In this case, the repackaged malware such as DroidKungFuSapp with assurance score V of 85.7% will be detected.

A more advanced countermeasure is to separate and identify the original benign portion of the app and the injected malicious code. In any repackaged app, the malicious components are highly communicated/connected together and loosely connected with other benign components. Hence, one possible way to identify this is to analyze the connectivity of the call graph of a repackaged app to identify the loosely connected or disconnected graph components. Then, one can compute features separately for each graph components and observe the imbalance. Table 2 shows the results of our assurance scores V for the benign and malicious components separately for some of the repackaged malware apps. The V scores for the benign components are much higher than the malicious components which show the validity of our proposed feature.

Table 2: Assurance scores for the benign and malicious components in some repackaged malware apps.

Repackaged Malware Name	**Assurance Score of Benign Component**	**Assurance Score of Malicious Component**
com.noisysounds	90%	26%
com.miniarmy.engine	100%	35%
com.chenyx.tiltmazs	78%	20%
com.craigsrace.headtoheadrcing	86%	28%

5.3. Free Popular Apps

Because the ground truth on trustworthiness of the free popular apps are not known, our analysis on them is more complex. Some of the classification decisions are validated through significant manual inspection of the code. We present our results on the *i)* assurance score computation, *ii)* classifications using two rules, and *iii)* new malware discovery.

5.3.1. Assurance Scores of Free Apps

Among the 2,684 free popular apps, 80.5% of them have 100% assurance scores, indicating that all the call sites of all the sensitive operations have valid user-trigger dependence. The detailed distribution of the assurance scores are shown in Figure 6. For the 80.5% of the apps that have 100% assurance scores, we utilized a signature-based malware scanning tool `VirusTotal` for additional validation. VirusTotal has 48 signature-based scanners (e.g., McAfee, NOD32, BitDefender). We found that only one scanner out of 48 scanners in `VirusTotal` triggers an adware alert for 13 free popular apps which have 100% assurance

scores (true positives). The rest of the free popular apps with 100% assurance scores are benign (true negatives), none of them trigger any alert by `VirusTotal`.

Through manual inspection, we find that the use of advertisement and analytics libraries is one main reason for sensitive operations to be called without proper user triggers. We selected several apps with less-than-100% V scores and computed their assurance scores with and without the ad/analytics libraries. The V scores are boosted significantly without the ad/analytics libraries. The results are shown in Table 3.

Table 3: Assurance scores of subset of selected benign apps including or excluding the ads/analytics libraries.

App Name	**Including Ads Libs**	**Excluding Ads Libs**
com.canadadroid.fantasy	75.0%	100.0%
com.canadadroid.penguinskiing	79.2%	100.0%
com.CalcFinalProgress	85.2%	96.3%
AzureNightwalker.ContactList	89.7%	97.4%

We also found a few malicious apps with high enough assurance scores (e.g., V is 89%) to pass our classification threshold (i.e., false negative), e.g., a spyware wallpaper app `com.ysler.wps.d3d` available on Google Play market.

5.3.2. Classification Results of Free Popular Apps

Our classification results are summarized in Table 4. Most of these free popular apps from Google Play market are classified as *benign* by both rules. Rule 1 labels 7.2% (193) of the 2,684 apps as malicious. We then applied Rule 2 to both categories of apps.

For apps classified as malicious by Rule 1. We applied Rule 2 to these 7.2% of the apps. Rule 2 labels 6.5% of the total (175 of 193) as malicious. The other 0.7% (18) are labeled benign.

For apps classified as benign by Rule 1. We applied Rule 2 to these 92.8% of the apps. Rule 2 labels 1.7% (47) of them as malicious, and classifies the rest 91.1% as benign.

There are 240 apps that are labeled as malicious by both *or* either one of the rules. Their popularity distribution is as follows, with higher concentrations of

<table>
<tr><td colspan="4">Rule 1 (V)</td></tr>
<tr><td colspan="2">Malicious</td><td colspan="2">Benign</td></tr>
<tr><td colspan="2">7.2%</td><td colspan="2">92.8%</td></tr>
<tr><td colspan="2">Rule 2 (DPVC)</td><td colspan="2">Rule 2 (DPVC)</td></tr>
<tr><td>Malicious</td><td>Benign</td><td>Malicious</td><td>Benign</td></tr>
<tr><td>6.5%</td><td>0.7%</td><td>1.7%</td><td>91.1%</td></tr>
</table>

Table 4: Summary of classification results after applying both rules on 2,684 free popular apps.

suspicious apps in medium and low popularity categories.

- High popularity category: 70 apps (29.2%)
- Medium popularity category: 87 apps (36.3%)
- Low popularity category: 83 apps (34.5%)

To confirm the correctness of our results, we then performed various code inspection on them, the detail of which are described next.

5.3.3. New Malicious Apps Found

To confirm that the apps classified as malicious are truly malicious, manual code inspection was performed. We also utilized the `VirusTotal` for additional validation.

Our method discovered many new malicious Android apps that *cannot* be detected by the VirusTotal tool [6]. These new malware apps did not trigger any alerts in VirusTotal. A subset of these new malicious apps is shown in Table 6 with examples of their sensitive function calls that lack of valid user-trigger dependence. All of them are confirmed by our manual analysis to have malicious functionalities. In Table 6, each column is a category of malicious action, e.g., unjustified dynamic code loading and unnecessary accessing of user information. Names of call sites without valid user-trigger dependence are given. All the apps shown in this table fail both of our classification rules, yet do not trigger any alerts in VirusTotal.

We highlight a few of the new malware that we discovered in the free popular apps. Our method detects a malicious app *Time Machine*, which is repackaged from an ebook app. The malware invokes many sensitive APIs (in `Jslibs` library) to perform unjustified operations, such as recording sound, retrieving phone state, and exfiltrating geolocation information. We find that an organizer app `com.via3apps.usobesit618` is bundled with a piece of malware collecting private information, such as device ID, email address, latitude and longitude, phone number, and username, and it uploads the details to a remote server. Another malware app is a game-guide app `com.bfrs.krokr`, which is bundled with adware `AndroidApperhand` (aka `Android.Counterclank`). `AndroidApperhand` is a piece of aggressive adware. It attempts to modify the browser's home page, copy bookmarks on the device, shortcuts, push notifications, and steal build information (brand, device, manufacturer, model). This adware also attempts to connect to a remote host.

For the apps that are labeled as malicious by *only one rule* (2.4% out of 2,684 apps), we have confirmed that most of the apps (2.2% of 2.4%) contain aggressive advertisement libraries, such as `Mobclix`, `Tapjoy`, and `Waps`. These libraries invoke sensitive operations without any user triggers. Unlike regular ad libraries, these aggressive ad libraries contain an *overwhelming* amount of

[6]Out of the 240 apps, 137 apps triggers at least one alert in VirusTotal.

invalid call sites. Most of them have a large number ($> 50\%$) of sensitive operations with zero valid call sites, which is consistent with known malware. Other researchers have also confirmed the potential security issues raised by these aggressive ad libraries Grace et al. (2012a).

5.3.4. False Positive Rate (FPR)

FPR is computed as $\frac{FP}{FP+TN}$, where TN stands for true negative (benign apps). 240 apps are classified as malicious by our method. VirusTotal scanning confirms 137 of them are malicious. For the rest of 103 apps, we randomly selected 21 apps out of these 103 apps and perform a thorough manually code inspection. We found that 11 of the 21 apps have definitive malicious or aggressive code behaviors that threaten the system assurance and data confidentiality in Android (described in Section 5.3.3 and Table 6). These behaviors were found in either the main components or adware. In the other 10 apps we did not find any threats, thus concluded that they are benign (false positives). The total false positives are estimated at $103 * \frac{10}{21} = 49$. Since the trustworthiness of the free popular apps is unknown, we used `VirusTotal` to check all the free popular apps classified as benign by our method (true negatives). We found that only one scanner out of 48 scanners in `VirusTotal` triggers an adware alert for 27 apps (true positives). The true negatives (TN) are $2684 - 240 + 49 - 27 = 2466$, yielding a 2.0% FPR.

5.4. Performance Evaluation

The experiments were conducted on a computer which has 3.0GHz Intel Core 2 Duo CPU E8400 processor and 3GB of RAM. We measure the time for parsing the AndroidManifest.xml file, Soot execution for constructing the dependence graph, the reachability analysis, and finding the dependence paths by traversing the graph. The average processing time for an app is about 158.01 seconds. This processing time does not include the time required to convert the dex format to jar. Table 5 shows the average time required by each analysis phase.

Table 5: Average feature-extraction time for an app.

Procedure	**Average Time (sec)**
Reachability Analysis	14.17
Finding Dependence Paths	54.30
AndroidManifest.xml Parsing	0.01
Graph Construction using Soot	89.53
Total Time	158.01

5.5. Summary

These experimental results suggest that our rule-based classification with a single complex feature is quite effective. We summarize our major experimental findings.

1. There are an overwhelming number of malware apps with zero or low assurance scores, indicating that a large portion of sensitive call sites in these programs are invalid.
   The DPVC vectors (representing a fine-grained distribution of per-operation valid call sites) of malware and benign apps have significantly different distributions (shown in Figure 5). Malware has a high concentration of zero or low per-operation valid call sites.
2. We obtained a low false negative (i.e., missed detection) rate of 2.1% when classifying 1,433 known malware samples based on their assurance scores and DPVC vectors, suggesting the effectiveness of our detection.
3. Our method identified 240 free popular apps (8.9%) as suspicious from Google Play market [7]. These malware excessively access device resources and personal information without any user knowledge. Our program analysis method effectively pinpoints these problematic call sites.
   Our method detects many malware that cannot be detected by VirusTotal scanning. Some of them are shown in Table 6. We confirmed them by manual code inspection. Our false positive rate ($\frac{FP}{FP+TN}$) is estimated at 2.0%.
   Our method identified more suspicious apps from the medium and low popularity categories than the high popularity category.
4. We observed several types of triggerless operations that are benign. Sensitive operations during *i)* app launching activities (e.g., `default_app_set.main.ver1`), *ii)* background service components (e.g., `com.monotype.android.font.dev.comic`), or *iii)* benign ad/analytical libraries (e.g., `rappsd.v1`) are typically automatically completed without user triggers.
   These factors result in lower assurance scores and skewed DPVC vectors, which may cause false positives. The classification accuracy is also affected by the accuracy of *Dare* in translating Dalvik bytecode to Java bytecode.

6. Discussion

In this section, we discuss the security guarantees provided by our app classification work, and sources of inaccuracy in our program analysis. We also describe possible extensions to the feature definitions.

[7]Google later took some malware apps off the Play market, e.g., `Us-Obesity-And-You-Teenagers`.

6.1. Security Analysis

Our app classification can be used to detect malware that invokes sensitive operations. Sensitive operations typically involve accessing system resources and sensitive data. Inferring their user-intention dependences enables the detection of potential data confidentiality and authorization issues. Examples of malicious patterns that can be detected by our analysis include:

- *Resource access:* executing sensitive operations without proper user triggers, such as sending unauthorized SMS messages, subscribing to premium-rate services automatically, or granting unauthorized root privilege to apps.
- *Data access:* accessing sensitive data items without proper user triggers, such as recording and uploading the victim's current location. Our static analysis does not track sensitive data variables. Instead, the function calls that may be used to access sensitive data are labeled (as operations) and analyzed.

In our model, the accuracy of the analysis is closely related to the accuracy of the data dependence analysis. Intra-procedural analysis captures fine-grained def-use relations within a function. The intra-procedural def-use relations can prevent a superfluous user input attack, thusly. One possible attack scenario is where the malware may require superfluous user inputs (before making function calls to conduct unauthorized activities) attempting to satisfy the dependence, but the user inputs are not consumed by the calls. For example, the user enters a phone number and a message to send SMS. The phone number entered by the user can be ignored and replaced with other number inside `sendTextMessage()` function. This type of data flow can be detected by tracking the dependence between the user inputs entered and the sensitive API calls, thus the superfluous user inputs can be identified.

Social engineering app is an application that provides fake user interface to look legitimate in order to circumvent the user and perform malicious activities (e.g., stealing money). Social engineering apps may demonstrate proper trigger-operation dependences, because of the seemingly conforming dependence paths between user triggers and sensitive operations. Therefore, due to the intrinsic nature of our user-intention analysis, it is not suitable for detecting social engineering apps. Possible solutions for this could be using app certification and user education.

The legitimate apps which require few user interactions may raise false alarm. For example, a calendar app can send an automatic reminder email message of a calendar event that previously scheduled by the user. Hence, the sensitive API that sends the email message may raise an alarm according to our security model since it is not explicitly triggered by the user. For example, the user has previously entered this event into the calendar. This action can be used as a trigger that justifies the operation of sending reminder emails. Our approach can be extended to address this problem by expanding and generalizing the definition of user triggers. The analysis for this calendar problem will be more complicated

than our current solution. The reason is that the information entered by the user is stored in a data structure or file to be read back when it is needed. Hence, there is no direct dependence between sending reminder email operation and the original user triggers used to store the information. One needs to expand and include this type of indirect dependence relation.

For the rule-based method, it is easy for the malware writers to game with the analysis than the machine learning-based classification. This is because the machine learning techniques utilize a large number of features compared to the rule-based method. So, it is harder for the attacker to compromise since she/he has to deal with many features in order to circumvent the security solution. On the other hand, the rule-based method might be easy for the attacker to compromise since she/he has to deal with a one/fewer number of features.

Precisely modeling a program's semantics and intention is in general challenging and open problem. In the seminal work on computer virus Cohen (1987), Cohen described the seminal impossibility result on malware analysis. The defense is still an open problem and similar arm-race issue exists in virtually all security solutions.

6.2. Sources of Inaccuracy in Feature Extraction

Overestimation of trigger-operation dependence may cause false negatives in the analysis report (i.e., failing to detect potentially malicious operations in the app). Certain dependence paths may only exist under specific data or control conditions. These branch conditions may not be statically predictable, resulting in overestimation. Some data dependence overestimation may be mitigated by identifying the specific conditions for certain dependence paths to be valid (e.g., by symbolic execution).

Conversely, underestimation of triggers may cause false positives. For instance, legitimate API calls can be triggered by runtime events such as clock-driven events from the calendar (e.g., the calendar app sends a reminder email message of a calendar event), or triggered by incoming network events. These runtime events may not be explicitly triggered by the user and thus lack the proper dependence according to our security model. One mitigation to the problem is to generalize and expand our definitions of triggers to include other legitimate triggering events. However, because triggers may be generated at runtime, static analysis alone may not be sufficient for feature extraction. Hybrid features extracted from both static and dynamic analyses are needed for complete dependence properties in a program. Its realization remains an interesting open problem.

Static program analysis has difficulty in performing the analysis on programs that employs obfuscation or encryption techniques. Obfuscation is mainly used to make the programs code difficult to understand.

Some Android apps use obfuscation to protect intellectual property Enck et al. (2011). ProGuard [8] is a recommended obfuscation tool by Google to pro-

[8] http://proguard.sourceforge.net/

tect against readability and does not obfuscate control flow. Hence, its impact is limited on static program analysis.

As indicated by Enck et al. (2011), it is easy to recognize some forms of the obfuscated code in Android apps. In particular, class, method, variable, and Java filename names are converted to single letters (e.g., a.java). However, several ads and analytics libraries are obfuscated to protect their intellectual property Enck et al. (2011). To obtain a rough estimate of the number of apps whose main code is obfuscated not the ads or analytics libraries, we used the same approach proposed in Enck et al. (2011) to search for a single letter Java filename (e.g., a.java) within a file path of the package name. This heuristic is used to obtain insight for finding obfuscation code in apps, but it is not a solid characterization. We found only 40 malware apps (2.8%) out of the 1433 apps have this code obfuscation. Moreover, we found 250 free popular apps (9.3%) out of the 2684 apps have this code obfuscation in part of their main code. Hence, we can infer from this statistics that the majority of the apps do not heavily employ code obfuscation. We applied our analysis on the reversed engineering Java bytecode using Dare tool to translate Dalivk bytecode to Java bytecode. The accuracy of our analysis is constrained by the accuracy of the reverse engineering tools.

There are several obfuscation techniques:

- Renaming technique: it renames classes, variables, and methods using meaningless names. This type of technique can not affect our approach since it just renames classes, variables, and methods without changing the content or the control flow structure.
- String encryption technique: it encrypts the string data.
- Control flow obfuscation technique: it reorders the code and inserts additional code statements while preserving the code semantic.

The latter two techniques can affect our approach since they change the data and the structure of the program. On possible solution is to use dynamic analysis Newsome and Song (2005); Yin et al. (2007) to provide insights about the programs runtime execution. As a future work, we plan to utilize the dynamic analysis with our user trigger dependence approach to get insights on which sensitive APIs are triggered by user inputs/actions. On way to do this is to label the user inputs/actions and to interpose the sensitive APIs in .apk file and insert monitoring code to get the sensitive API call logs during the app execution.

7. Related Work

We categorize related Android app analysis work into *i)* classification with static features and *ii)* classification with dynamic or hybrid features. Both approaches are necessary for evaluating app security, providing complementary

behavioral profiles [9]. We compare some of the existing mobile app classification solutions in Table 7.

Classification with static features. In order to infer the trustworthiness of mobile applications, multiple approaches have been proposed to statically extract properties of a program from its code and/or its requested permissions (e.g., Peng et al. (2012); Sanz et al. (2012)). One of the earliest such work is SCanDroid Fuchs et al. (2009). SCanDroid Fuchs et al. (2009) proposed to extract security specifications from the app's manifest and check whether the data-flows through the app are consistent with the stated specifications. [10]

The solution by Peng et al. (2012) calculated risk scores from the permissions requested by Android apps and found the hierarchical mixture of naive Bayes to be the best classifier for the risk score based app classification. The work by Sanz et al. (2012) also extracted permission-usage based features, and evaluated several classifiers including random forests, naive Bayes, and Bayesian network. The false positive rate in Sanz et al. (2012) is higher than 11%.

DroidAPIMiner Aafer et al. (2013) extracted features related to API calls, and evaluated several machine learning classifiers including k-nearest neighbor (KNN), decision tree, and support vector machines. It achieves a 97.8% detection rate of the malware samples and a false positive rate of 2.2% with KNN. Drebin Arp et al. (2014) analyzed `AndroidManifest.xml` and disassembled code to extract features on requested permissions and API calls, and used support vector machines (SVM) as a classifier. Drebin achieves 94% detection rate of the malware samples at a false positive rate of 1%. Both work used multiple sets of features as opposed to our work. A recent paper Wolfe et al. (2014) on Android malware classification utilizes the assurance score feature and dozens of other manifest-based features. The solution by Wolfe et al. (2014) achieves similar accuracy as ours. It utilizes a significant number of features than our work. It employs machine learning techniques, as opposed to our simple rule-based classification.

In comparison to the above permission-based classification, features extracted from code analysis are more fine-grained and specific. We highlight several such solutions next. The security goal in AndroidLeaks Gibler et al. (2012), SCANDAL Kim et al. (2012), and PiOS Egele et al. (2011) for iOS is focused on detecting data leak vulnerabilities, specifically on information flow for confidentiality analysis. The methods label sensitive data/sources and potentially risky sinks (typically network API calls) and report when there are data-leaking dependence paths between them. PiOS reports a 13% false negative rate.

Although using dependence-path based analysis, our definitions for the path have different semantics. As a result, our analysis with a complete coverage of sensitive operations provides comprehensive app profiling, which offers more protection than data confidentiality. For example, our analysis also detects system-

[9]Not all related papers report both FP and FN rates.
[10]No experimental results were reported in SCanDroid.

assurance-related operations such as unauthorized camera access or recording, which is out of the scope the data leak solutions.

Multiple features were utilized to make classification decisions in RiskRanker Grace et al. (2012b). The classification is based on several types of suspicious behavior signatures extracted through control-flow and intra-method data-flow analyses. An example of such suspicious behaviors include accessing sensitive data in a dependence path that also contains decryption (usually for deobfuscation) and execution methods. RiskRanker reports a 9% false negative rate. In comparison, our method enforces benign properties of trustworthy programs (as opposed to detecting malicious properties). Our results also show better classification accuracy compared to the existing approaches.

DroidSIFT Zhang et al. (2014) is a recent Android malware classification system that is based on constructing dependence graphs to model the dependences between API calls. Its feature vector is extracted from the graphs. The work built graph databases for known benign and malicious Android apps, and performed graph similarity queries (based on graph edit distance) for unknown apps. Its approach correctly classifies 93% of known malware samples (with naive Bayes classifier). Their anomaly detector based on the benign graph database achieves a false negative rate of 2% and a false positive rate of 5.15%. The semantics of dependence properties in our work and DroidSIFT are different. Our work models the data dependency between user-input functions and sensitive APIs. Consequently, the classification mechanisms are different. Our solution – based on rules – does not rely on graph similarity computation, which might be expensive for large graphs.

Classification with dynamic or hybrid features. Solutions in this category detect malware apps by their runtime execution patterns (i.e., dynamic features), sometimes together with statically extracted features. Andromaly Shabtai et al. (2012) and Amos et al. (2013) extract dynamic features including memory activity and CPU load to classify Android apps. They apply several classifiers including decision trees, naive Bayes, and Bayesian networks. The best classifier in Andromaly Shabtai et al. (2012) achieves a 10.4% false positive rate. In Amos et al. (2013) the false positive rate is over 15%. The work by Liu et al. (2009) detected malicious behaviors on mobile devices by monitoring abnormal power consumption due to malware activities, and reports a false positive rate that ranges from 4.3% to 10%.

Crowdroid Burguera et al. (2011) performs k-means clustering algorithms on dynamic features collected from Android apps. The features are the frequencies of occurrences for system calls (e.g., `open()`, `kill()`) executed by an app. The proposed solution successfully identifies all of the author-created malware, while it reports a 20% false positive rate on the real-world repackaged malware.

The features in DroidRanger Zhou et al. (2012) are hybrid. It statically extracts behavioral signatures of known malware samples. Examples of static features include sequences of APIs being called, package names, and class hierarchies. It also has a dynamic execution monitor that inspects the suspicious runtime behaviors of the app, such as loading dynamic code. The method reports a false negative rate of 4.2%.

These dynamic analysis provides useful information on runtime program behaviors and complements our static analysis work. Both approaches are necessary for app classification.

Non-classification work. Several validation and verification solutions have been proposed for mobile platforms to enhance the assurance of execution. These tools gather contextual information associated with sensitive operation invocations. This information is compared with models built through hybrid program analysis. For example, AppIntent Yang et al. (2013) defines privacy leakage as user-unintended data transmission. It provides a security analyst the context information associated with the transmission. The human analyst then decides whether the transmission is legitimate or not. Pegasus Chen et al. (2013) proposes a Permission Event Graph abstraction in order to detect sensitive operation invocations that are inconsistent with the UI events. It automatically verifies the app's behaviors with respect to pre-defined app-specific policies. CHEX Lu et al. (2012) identifies potentially vulnerable component interfaces that are exposed to the public without proper access restrictions in Android apps. The analysis detects apps that are vulnerable, but not necessarily malicious. The authors utilized data-flow based reachability analysis. CHEX reports a false positive rate of 19%. ComDroid Chin et al. (2011) characterizes security vulnerabilities caused by Android inter-app communication. User-driven access control gadget (ACG) was proposed in Roesner et al. (2012) to capture user authorization actions (keyboard shortcut or mouse movement) for assured resource access at runtime. Unlike ours, these solutions are not for malware classification, thus have different security goals and technical approaches from ours.

8. Conclusions and Future Work

We demonstrated the high classification accuracy achieved by using a single well-prepared feature on Java programs. What differs our feature from those used in existing work is that our classification enforces carefully-chosen benign properties in programs. These benign properties are observed in trustworthy programs, but not in malware. Our enforcement of these benign properties through mobile app classification allows defenders to stay ahead of the game in the eternal armrace between attack and defense Cohen (1987).

For future work, we plan to generalize the dependence definitions to include non-user triggers. We also plan to utilize advanced program analysis techniques to further improve the classification accuracy. For the deployment perspective, we plan to provide and present informative and intuitive interpretation of the multiple dimensional analysis results from various tools to users.

9. References

Soot: a Java optimization framework. `http://www.sable.mcgill.ca/soot/`.

Yousra Aafer, Wenliang Du, and Heng Yin. DroidAPIMiner: Mining API-level features for robust malware detection in Android. In *Proc. of 9th International Security and Privacy in Communication Networks (SecureComm)*, pages 86–103, 2013.

Brandon Amos, Hamilton A. Turner, and Jules White. Applying machine learning classifiers to dynamic Android malware detection at scale. In *Proc. of 9th the International Wireless Communications and Mobile Computing Conference, IWCMC*, pages 1666–1671. IEEE, 2013.

Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. Drebin: Efficient and explainable detection of Android malware in your pocket. In *Proc. of 17th Network and Distributed System Security Symposium (NDSS)*, 2014.

Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for Android. In *Proc. of the 1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM)*, pages 15–26. ACM, 2011.

Kevin Zhijie Chen, Noah M. Johnson, Vijay D'Silva, Shuaifu Dai, Kyle MacNamara, Tom Magrino, Edward XueJun Wu, Martin Rinard, and Dawn Song. Contextual policy enforcement in Android applications with permission event graphs. In *20th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2013.

Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *Proc. of the 9th Int'l Conference on Mobile Systems, Applications, and Services*, pages 239–252. ACM, 2011.

F. Cohen. Computer viruses theory and experiments. *Computers and Security*, 6:22 – 35, 1987.

Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the clones: Detecting cloned applications on Android markets. In *Proc. of the 17th European Symposium on Research in Computer Security (ESORICS)*, volume 7459 of *Lecture Notes in Computer Science*, pages 37–54. Springer, 2012.

Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on Android. In *Proc. of the 13th International Conference on Information Security (ISC)*, pages 346–360. Springer-Verlag, 2010.

Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proce. of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2011.

Karim O. Elish, Danfeng Yao, and Barbara G. Ryder. User-centric dependence analysis for identifying malicious mobile apps. In *Proc. of the IEEE Mobile Security Technologies (MoST) workshop, in conjunction with the IEEE Symposium on Security and Privacy*, 2012.

William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, pages 393–407. USENIX Association, 2010.

William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of Android application security. In *Proc. of the 20th USENIX conference on Security*. USENIX Association, 2011.

forti-guard. Fortinet FortiGuard Labs Reports. August 2013. `https://www.fortinet.com`.

Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. SCanDroid: Automated security certification of Android applications, 2009. Technical report, University of Maryland.

Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *Proc. of the 5th International Conference on Trust & Trustworthy Computing (TRUST)*, volume 7344 of *Lecture Notes in Computer Science*, pages 291–307. Springer, 2012.

Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proc. of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WISEC)*, pages 101–112. ACM, 2012a.

Michael C. Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. RiskRanker: scalable and accurate zero-day Android malware detection. In *Proc. of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 281–294. ACM, 2012b.

Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12:26–60, 1990.

Jinyung Kim, Yongho Yoon, Kwangkeun Yi, and Junbum Shin. SCANDAL: Static analyzer for detecting privacy leaks in android applications. In *Proc. of the IEEE Mobile Security Technologies (MoST) workshop, in conjunction with the IEEE Symposium on Security and Privacy*, 2012.

Lei Liu, Guanhua Yan, Xinwen Zhang, and Songqing Chen. VirusMeter: Preventing your cellphone from spies. In *Proc. of the 12th International Symposium on Recent Advances in Intrusion Detection*, pages 244–264. Springer, 2009.

Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: statically vetting Android apps for component hijacking vulnerabilities. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, pages 229–240. ACM, 2012.

James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of the Network and Distributed System Security Symposium*. The Internet Society, 2005.

Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting Android applications to Java bytecode. In *Proc. of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2012.

Hao Peng, Chris Gates, Bhaskar Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Using probabilistic generative models for ranking risks of Android apps. In *Proc. of the ACM conference on Computer and Communications Security (CCS)*, pages 241–252. ACM, 2012.

Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J. Wang, and Crispin Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 224–238, 2012.

B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P.G. Bringas, and G. Alvarez. Puma: Permission usage to detect malware in Android. In *Proc. of International Joint Conference CISIS'12-ICEUTE'12-SOCO'12 Special Sessions*, 2012.

Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. Andromaly: a behavioral malware detection framework for Android devices. *Journal of Intelligent Information Systems*, 38(1):161–190, 2012.

Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley, 2006.

Britton Wolfe, Karim Elish, and Danfeng Yao. Comprehensive behavior profiling for proactive Android malware detection. In *Proc. of 17th International Information Security Conference (ISC)*, 2014.

Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. AppIntent: Analyzing sensitive data transmission in Android for privacy leakage detection. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2013.

Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, pages 116–127. ACM, 2007.

Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware Android malware classification using weighted contextual API dependency graphs. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS'14)*, November 2014.

Yajin Zhou and Xuxian Jiang. Dissecting Android malware: Characterization and evolution. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 95–109, 2012.

Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Proc. of the 19th Network and Distributed System Security Symposium (NDSS)*, 2012.

Table 6: Malicious activities of a subset of new malware found by our method.

App Name	Access IMEI/Device Info	Access User ID	Access Geolocation	Access IP Address	Access Bookmark	Load Code Dynamically	Send SMS
com.pougamefree.cheats[†]	getDeviceId()	-	getLongitude() getLatitude()	-	getAllBookmarks()	-	-
com.canny.FishHunter	getDeviceId() getLine1Number() getSimSerialNumber() getSubscriberId()	-	getLongitude() getLatitude() getLastKnownLocation() getAccuracy()	-	getAllBookmarks()	-	-
			getLatitude() getLastKnownLocation()				
com.hd.peliculashd[†]	getDeviceId()	-	getLongitude() getLatitude() getLastKnownLocation() getAccuracy()	getIpAddress()	-	-	-
oms.cj.kobodl	getDeviceId() getLine1Number() getSimSerialNumber()	-	getLongitude() getLatitude()	-	-	-	-
com.canny.RankSwap	getDeviceId() getLine1Number() getSimSerialNumber() getSubscriberId()	-	getLongitude() getLatitude() getLastKnownLocation() getAccuracy()	-	getAllBookmarks()	-	-
com.via3apps.relation958	getDeviceId()	-	getLongitude() getLatitude() getLastKnownLocation()	-	-	-	-
com.berobo.android.scanner	getDeviceId() getLine1Number()	getAccounts()	getLongitude() getLatitude() getLastKnownLocation() getAccuracy()	getIpAddress()	-	loadClass()	sendTextMessage()
com.Amazing***BibleFree	-	-	getLongitude() getLatitude() getAccuracy()	-	-	-	-

[†] App has been removed from Google play market by 12/05/2013.

Table 7: Comparison with related mobile app classification work.

Solution	Aim	Feature Type	# Features*	Feature Category**	Classification Policy/Algorithm	Evaluation Scale	Apps Collected From	Classification Accuracy
AndroidLeaks Gibler et al. (2012)	Confidentiality	Static	S	DS	Rule: sensitive data used by risky APIs	24,350 apps	various Android markets	FP = 35%
Crowdroid Burguera et al. (2011)	Malware classification	Dynamic	M	DI	k-means clustering	3 self-written malware and 2 real malware	VirusTotal	FP = 20%
Amos et al. (2013)	Malware classification	Dynamic	M	DI	naive Bayes, Bayes nets, MLP, logistic regression, RF, DT	training (408 benign, 1,330 malware), testing (24 benign, 23 malware)	Android malware genome project, VirusTotal, Google Play	FP = 15%
PiOS Egele et al. (2011)	Confidentiality	Static	S	DS	Rule: sensitive data used by risky APIs	1,407 apps	Apple's iTunes, BigBoss	FN = 13%
Sanz et al. (2012)	Malware classification	Static	M	DI	logistic regression, naive Bayes, Bayes nets, SVM, KNN, DT, RF	1,811 benign, 249 malware	Google Play, VirusTotal	FP = 11%
Andromaly Shabtai et al. (2012)	Malware classification	Dynamic	M	DI	naive Bayes, Bayes nets, histograms, k-means, LR, DT	4 self-written malware, 40 benign	Google Play	FP = 10%
RiskRanker Grace et al. (2012b)	Detection of abnormal code/behavior patterns	Static	M	DS	Rule: multiple malware behavior signatures	118,318 apps	various Android markets	FN = 9%
Peng et al. (2012)	Risk assessment	Static	M	DI	different probabilistic generative models	model generation & testing (71,331 apps) validation (136,534 apps), 378 malware	Google Play, Android malware genome project	FP = 4%
DroidAPIMiner Aafer et al. (2013)	Malware classification	Static	M	DI	KNN, DT, SVM	16,000 benign, 3,987 malware	Google Play, Android malware genome project, McAfee	FP = 2.2% FN = 2.2%
Drebin Arp et al. (2014)	Malware classification	Static	M	DI	SVM	123,453 benign, 5,560 malware	various markets, malware forums and security blogs, Android malware genome project	FP = 1% FN = 6%
Ours	Identification of unauthorized API calls	Static	S	DS	Rule: trigger-based dependence for privileged API calls	2,684 benign, 1,433 malware	Google Play, VirusShare, Android malware genome project	FP = 2% FN = 2.1%

*Number of features: single feature (S) or multiple features (M).

**Feature category: domain-specific (DS) or domain-independent (DI).