A Static Assurance Analysis of Android Applications

Karim O. Elish*, Danfeng (Daphne) Yao*, Barbara G. Ryder*, and Xuxian Jiang**

*Department of Computer Science, Virginia Tech
{kelish, danfeng, ryder}@cs.vt.edu

**Department of Computer Science, North Carolina State University
{jiang}@cs.ncsu.edu

Abstract

We describe an efficient approach to identify malicious Android applications through specialized static program analysis. Our solution – referred to as *user-intention program dependence analysis* – performs offline analysis to find the dependence relations between user triggers and entry points to methods providing critical system functions. Analyzing these types of dependences in programs can identify the privileged operations (e.g., file, network operations and sensitive data access) that are not intended by users. We apply our technique on 708 free popular apps and 482 malware apps for Android OS, and the experimental results show that our technique can differentiate between legitimate and malware applications with high accuracy. We also explain the limitations of the user-intention-based approach and point out the need for practitioners to adopt multiple analysis tools for evaluating the assurance of Android applications.

1 Introduction

Malicious mobile apps and vulnerable mobile computing platforms threaten the privacy of personal data and device integrity. These threats have been extensively demonstrated in Android environments (e.g., [12, 18, 28]). Smartphones running malicious apps also increase the attack surface of organizational local area networks and may allow attackers to compromise stationary hosts and servers to which they are connected threatening data and systems of a much larger scale than mere individual nodes.

Because of the diversity of mobile apps, most of the existing approaches for achieving mobile computing security analyze, manage, and/or monitor mobile applications at installation time and runtime. These approaches include:

- installation-time permission mechanisms where users give explicit access authorizations to apps (available in Android by default),
- conventional virus scan approaches (e.g., VirusTotal),
- run-time monitoring of information flow (e.g., TaintDroid [18], user-driven permissions [46]),
- static program analysis where source code or binaries of apps are analyzed to estimate their behavior patterns (e.g., data flow analysis [25, 26, 28, 37] or control flow analysis [8]).

In this paper, we describe a new user-intention-based static analysis method for classifying applications. We first briefly review the static program analysis technique and then explain the novelty of our approach. The static program analysis solutions in the Android security paradigm specialize the classic information flow analysis techniques in the Android environment with various customized definitions of sinks and sources. Two main categories of threats in the analysis are *the leak of sensitive data* and *unauthorized access to system resources* (e.g., malicious apps sending spam) [1].

Confidentiality and authorization are two main security goals in the program analysis of apps. For example, SCanDroid [25] extracts security specifications from the manifest of an app and checks whether data flows through the app are consistent with the stated specifications. We compare existing static analysis solutions in Table 1 and describe them in detail in Section 6. These complementary static analysis tools and policies can be utilized to assess the trustworthiness of an application, a process sometimes referred to as certification [25] and vetting [37].

Software assurance refers to the degree of confidence that the software functions in the intended manner and is trustworthy [2, 38] (i.e., free from vulnerabilities). The assurance assessment of software applications demands multiple heterogeneous analysis mechanisms; there is no silver bullet.

[1] Thanks to JVM's safety properties, classic program vulnerabilities such as buffer overflow are usually excluded in the analysis.

Table 1: Comparison of select static analysis solutions for apps. UID refers to our user-intention dependence analysis.

Solution	Aim	Flow Analysis	Classification Policy	Evaluation Scale
SCanDroid [25]	Enforcement of confidentiality, integrity	Data, string	Constraints on permission logics	N/A
CHEX [37]	Discovery of exposed component API	Data	Component exported to public without restrictions	5,486 apps
RiskRanker [30]	Detection of abnormal code/behavior patterns	Data, control	Multiple malware behavior signatures	118,318 apps
Woodpecker [28]	Firmware permission	Data, control	N/A	8 phone images, 13 permissions
AndroidLeaks [26]	Confidentiality	Data	Sensitive data used by risky APIs	24,350 apps
SCANDAL [33]	Confidentiality	Data	Sensitive data used by risky APIs	90 apps & 8 malware
Stowaway [21]	Detection of overprivileged apps	String, Intent control flow	Compare required and requested permissions	940 apps
ComDroid [8]	Detection of apps communication vulnerabilities	Intent control flow	Implicit Intent with weak or no permission	100 apps
PiOS [17]	Confidentiality	Data	Sensitive data used by risky APIs	1,407 apps
UID	Identification of unauthorized calls	Data, event-specific control	Trigger-operation dependence for privileged function calls	708 apps & 482 malware

In this work, we describe a new analysis approach that leverages the dependence effects on program behaviors and attempts to understand the reasons and causes of operations. Smartphone apps (Android, iOS, or Windows Phone) are unique in their user-centered and interaction-intensive design, where operations may require users' specific actions to initiate. We ask how to utilize user intention in *automatic* and *scalable* program analysis for assurance evaluation? Such an analysis method needs to also cover inter-app communication (e.g., through Intent in Android), where user intention may trigger operations in multiple apps.

We present an approach that analyzes the dependence relations between user inputs/actions and entry points to methods providing critical system functions. Such an analysis – referred to by us as *user-intention program dependence analysis* – captures and enforces the causal relations in programs. Our *hypothesis* is that in benign apps, critical system operations such as network and file access are directly or indirectly initiated by users' requests, whereas mal-intended apps, such as spyware or Trojans, often abuse data and system resources without proper explicit user consent. If this hypothesis holds, dependence analysis between user triggers and operations on application code can contribute to automated assurance assessment of unknown apps. Our experimental results analyzing 708 benign apps and 482 malware apps provide positive evidence for the hypothesis and this user-intention program dependence analysis can provide useful app assurance assessment.

Notable recent papers sharing a similar user-intention-based hypothesis include user-driven access control [46, 48], and BINDER for run-time network assurance [11]. They are run-time policy-based monitoring solutions. For example, BINDER describes traffic dependence analysis and solves a completely different problem. We demonstrate the effectiveness of user-intention-based static program analysis. Our approach is fundamentally different from the existing user-driven access control solutions. One of the advantages of our approach is that it does not require any user participation, as the analysis is performed offline.

Our technical contributions are summarized as follows.

1. We designed an user-intention dependence analysis for identifying malicious Android applications. We introduce a specialized static program analysis for identifying the directed paths between user inputs (e.g., data or actions) and entry points to meth-

ods providing critical system services. The analysis produces a quantitative assurance score, which is the percentage of critical function calls that are triggered by user inputs. We also thoroughly discuss limitations and sources of inaccuracy in this approach.

2. We implemented a static analysis tool for the proposed dependence analysis for Android applications. Our tool provides specialized flow analysis through finding the required dependence paths that conform to our hypothesized application behavior model. The tool parses Java bytecode and constructs a context-sensitive data-flow dependence graph through intra- and inter-procedural def-use analysis, and event-based information flow that includes the handling of data flow through Android intents and GUI related implicit method invocations.
3. We evaluated our static analysis framework by conducting large-scale experiments (with 1,190 real-world Android apps) in order to characterize the behaviors of legitimate and malicious Android apps, specifically on how they respond to user inputs and events. Our results show encouraging evidence for the validity of our dependence hypothesis. The analysis successfully classifies most of 482 malicious apps (e.g., 1.5% false negative rate when the assurance score threshold is set to 80%, and 0.2% false negative rate when the threshold is set to 100%). The majority of 708 free popular apps from the Android app market conforms to our hypothesis; 91.5% of these apps give full 100% assurance score (i.e., every critical function call in the program has the required user dependence property). Among the free popular apps that do not have full assurance score, our tool pinpoints operations that leak sensitive information without user triggers.

The use of static analysis for program classification is not surprising. What is surprising is the effectiveness of the simple user-intention-based dependence analysis. The analysis attempts to investigate the dependences that affect program behaviors and interpret the motives of operations from the perspect of user intention. This logical interpretation of program behaviors is useful and new. It aims to enforce legitimate patterns in programs as opposed to identify malicious ones. Although the long-term effectiveness of this type of analyses needs further evaluation, we envision other similar static analysis solutions in the future that exploit high level semantic or logical features in legitimate program behaviors.

The rest of this paper is organized as follows. Section 2 presents an overview and definitions of our user-intention dependence analysis approach. Section 3 describes the design and details of our analysis method. Section 4 describes the implementation and provides the evaluation results. Section 5 discusses the security analysis of our approach. Finally, we review related work in Section 6 and outline the directions for future work in Section 7.

2 Overview and Definitions

Data flow and control flow dependence analyses are well known techniques in program analysis and information flow. Our goal is to customize them for Android app assurance assessment. We describe a specialized dependence analysis on Java bytecode where one traces back the initial triggers or causes of sensitive function calls (e.g., library calls for performing network and file system operations). Sensitive function calls lacking proper triggers are reported and may indicate unauthorized access to data or system resources by the app. The definition for proper triggers may vary according to the semantics of the analysis. We demonstrate in this work that having user intention as the trigger in the dependence analysis yields a unique approach and produces useful insights on app behaviors. User intention may be embodied in user inputs, actions, or events such as Android Intent. We call this approach *user-intention dependence analysis* for programs (or UID for short). Server-side software such as Apache web server or Sendmail email server typically lacks direct user interaction, as users interact with the servers through network requests. In comparison, modern smartphone apps are intensely user interactive, which makes this approach appropriate.

Our analysis computes an assurance score (defined in Equation 1) for a program that indicates the percentage of critical function call invocations in the program that are triggered by user inputs. In our model the higher the assurance score is, the more trustworthy the program. The analysis results help security experts with app classification (e.g., into different trustworthiness categories), and can be combined with other static program analysis reports (e.g., [25, 26, 28, 37]). Our experiments on 1,190 apps show that the analysis provides a strong indicator for identifying malicious apps.

We aim at exposing possible privileged actions of apps that are not intended by the user and lack proper dependences in the code. We define our terminology used in our user-intention approach as follows:

Trigger refers to a user's input or action to the app. A trigger is a variable defined in the program. For example, the user's input may be text (sequence of char string) entered via text field, while the user's action is any click on UI element such as button. We consider relevant API calls in UI objects that return an user's input value or listen to user's action, as triggers or sources.

Operation refers to an entry point of a method providing functionalities such as network I/O, file I/O, telephony services. An operation is a function call in the program. We consider relevant API calls that are related to send/receive network traffic, create/read/write/delete operations for files, insert/update/delete operations 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 as operations or sinks.

Dependence path from trigger to operation refers to a directed path that captures the dependence relations between a trigger and an operation in a program. There are two types of dependence semantics, data flow dependence and control flow dependence. The former specifies a definition-and-consumption (*def-use*) relation, where a trigger is defined and later used as an argument to an operation, and the latter specifies the function call sequences. The trigger may be transformed before being used as an argument in the operation, thus the dependence path between them may be long.

Both sets of triggers and operations need to be specified for the analysis. Their sizes affect the run-time and storage complexities.

Security goal of analysis is to prevent unauthorized privileged actions that are not intended by the user. Because the analysis is static (as opposed to run-time) without any user participation, user intention needs to be approximated. In our analysis user intention is embodied in the trigger variables.

We define the user-intention dependence-based assurance score in Equation 1, which is the percentage V of operations that have valid triggers on their dependence paths among all occurrences of operations. In our model, the assurance score $V \in [0\%, 100\%]$ represents the portion of critical function calls that are intended by the user; a high V value is desirable.

$$V = \frac{\text{\# of operations with triggers on dependence paths}}{\text{Total number of operations}} \tag{1}$$

A threshold T needs to be specified when classifying programs into benign or suspicious classes. For example, a security policy may be that if $V > T$, then the program is benign, otherwise suspicious.

A data dependence graph (DDG) is a common program analysis structure which represents interprocedural flows of data through a program [32]. 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 instruction I_i to instruction I_j with respect to a variable x in P.

Trigger propagation through events. Our analysis needs to track the propagation of triggers through events, specifically Intent, which is an event-based mechanism for the communication between applications or components in Android. Information entered by the user in one Activity may be passed to another Activity 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 on dependences related to user activities, we aim to produce Intent-specific control flow dependence analysis, as opposed to general control flow analysis.

3 Details of Our Analysis Methods

We aim to identify unauthorized actions of apps that are not intended by users. Our approach is to characterize programs by statically examining how many sensitive operations (data or system resource access) depend on some forms of user inputs. Mobile apps typically have intense user interaction, allowing us to approximate implicit user intention with explicit user inputs and actions. The workflow of our dependence analysis described in this section is illustrated in Figure 1. The algorithm performs the following high level steps. Pseudocode of our procedure is given in Algorithm 1.

(i) Specify operations We identify a set of sensitive or critical API calls to Android and Java libraries that may be used to perform useful operations such as data access, network and file I/O access as operations or sinks. This task is app-independent and does not need to be repeated for each app.

(ii) Specify triggers in the app We identify all possible entries where user inputs (i.e., data or actions) may be entered as triggers or sources.

(iii) Construct dependence graph Given the source code or bytecode of an app, the algorithm constructs the complete data dependence graph based on def-use relations. Based on the event labels, the graph is further augmented to cover the dependences through event-based propagation (e.g., through Intent or implicit method invocation).

(iv) Identify dependence paths The algorithm identifies

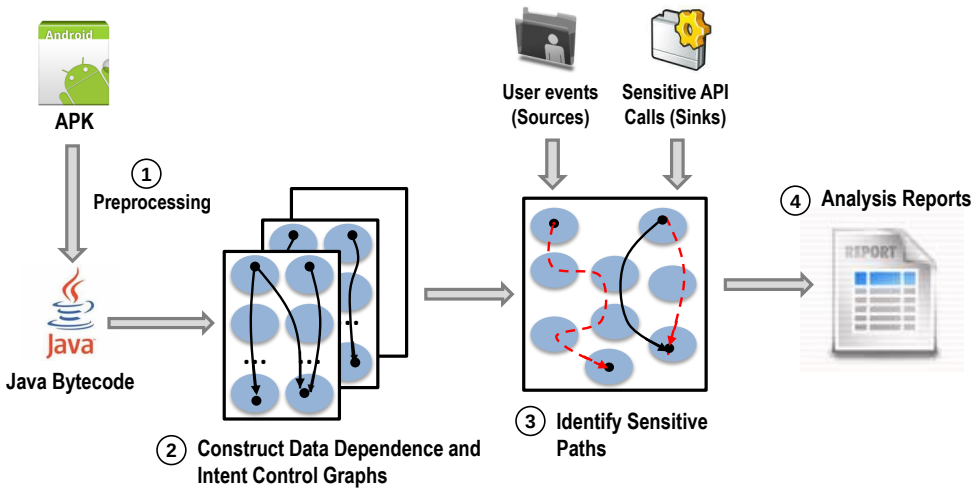


Figure 1: Workflow in our user-intention dependence analysis.

the data-dependence paths between triggers and operations through the augmented dependence graph. Specifically, for each occurrence of an operation in the app, it traverses the dependence graph to check for the existence of a path (i.e., whether or not there exists a dependence path from any of the triggers to this operation). If no path exists between the user inputs and the sensitive API call, this operation (API call) is reported as a risky/suspicious API call which needs further inspection.

In steps *(i)* and *(ii)*, we manually identify the initial trigger and operation sets. Then, scanning apps for the occurrences of triggers and operations in the sets is automated. Steps *(i)*, *(ii)*, and *(iii)* are independent of each other, thus the order of execution does not matter.

For each app, an assurance score representing the percentage of sensitive API calls which are not triggered by the user (in Equation 1) is computed and reported. Also reported are useful details and statistics such as the names and locations of these calls for further diagnosis.

3.1 Dependence Graph Construction

Our method for constructing the dependence graph based on explicit def-use relations is presented. Then we explain why and how it is augmented in order to capture def-use relations through events.

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, while the inter-procedural dependences edges are identified based on constructing a call-site context-sensitive call graph. The context-sensitive analysis considers the context in the caller function when analyzing a callee function. In particular, it differentiates between multiple function calls of the same function with respect to provided arguments. On the other hand, a context-insensitive analysis does not differentiate between multiple calls of the same function with different arguments. Thus, a context-insensitive analysis may not provide as accurate a solution.

Because events may be implicit, data flow associated with events is not handled by general-purpose data flow analysis techniques. Thus event-specific dependence analysis is required; otherwise the analysis results may underestimate the actual dependence relations, yielding false alarms.

Event-specific data dependence

We consider 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. The approach is to perform necessary control flow analysis, which finds *bridges* between disjoint graph components, so that one can obtain the complete reachability of trigger variables.

We describe our 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. An 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.

We give details of how our analysis handles explicit Intent, where the target component name is specified. Our analysis includes knowledge of the standard actions of the system built-in components to identify them (e.g., `android.intent.action.VIEW` action to launch map app and show location, and `android.intent.action.DIAL` action to launch phone app and dial a specific number).

We first identify the source component and the target component that are linked through an Intent object. The analysis identifies 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 target's name, to obtain the target component.

Then, based on the information obtained, 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. A static analysis approach for analyzing an implicit Intent is by processing the manifest 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. We address this problem by linking the dependent calls to the relevant API functions 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()`.

3.2 Finding Dependence Paths

Once the dependence graph is constructed, checking the path between a source and sink pair becomes straightforward. In Algorithm 1, `checkPathExistence()` performs this task. It checks if two nodes in the graph (N_{source} and N_{sink}) are connected by traversing the dependence graph (i.e., is there a path from node N_{source} to node N_{sink}). If no path exists, we report N_{sink} as a sensitive API call for further inspection. The resulting paths are used for calculating the assurance score.

In summary, our static analysis framework constructs a context-sensitive data-flow dependence graph with intra- and inter-procedural dependence analysis, and intra- and inter-application Intent-based dependence analysis. The graph construction method is general and the graphs may be used for other analyses beyond user dependence.

Algorithm 1 User-intention dependence analysis

Input: $A \leftarrow$ {App source/bytecode}
$U_i \leftarrow$ {Set of user triggers of an app}
$M_s \leftarrow$ {Set of sensitive operations of an app}
Output: $V \leftarrow$ {Assurance score}
$M_n \leftarrow$ {Set of sensitive operations not dependent on user triggers}

1: $G \leftarrow$ ConstructDependenceGraph(A)
2: **for** each $m_s \in M_s$ **do**
3: **for** each $u_i \in U_i$ **do**
4: checkPathExistence(u_i, m_s, G)
5: **end for**
6: **if** no path exists between $(u_i \wedge m_s) \in G$ **then**
7: $M_n \leftarrow M_n \cup \{m_s\}$
8: **end if**
9: **end for**
10: // L is list of operations triggered by user
11: $L \leftarrow M_s - M_n$
12:
13: $V \leftarrow \frac{\text{\# of operations in } L}{\text{\# of operations in } M_s}$
14:
15: **return** (V, M_n)
16:
17: **procedure** CHECKPATHEXISTENCE(u_i, m_s, G)
18: create queue Q
19: Q.enqueue(u_i)
20: **while** Q is not empty **do**
21: $n \leftarrow Q.dequeue()$
22: **if** n == m_s **then**
23: **return** True
24: **end if**
25: list N $\leftarrow$ get all direct successors nodes of u_i
26: **for** each $node \in N$ **do**
27: **if** *node* not visited before **then**
28: Q.enqueue(*node*)
29: **end if**
30: **end for**
31: **end while**
32: **return** False
33: **end procedure**

4 Experimental Evaluation

We implemented our static analysis framework based on Soot (a static analysis toolkit for Java) [3]. Soot transforms Java bytecode into an intermediate code representation suitable for analysis. We also utilized the def-

use structures provided by Soot. Soot does not provide inter-procedural call information. Thus, we implemented our own analysis method to augment the def-use relations across the boundaries of methods. We identified a list of sensitive API calls to be used for identifying sensitive operations in each app. As described in Section 2, those operations can utilize system resources, such as network I/O, file I/O, GSM-specific telephony services, (e.g., `sendTextMessage()` provided by `android.telephony.gsm.SmsManager` library for sending SMS, and `openFileOutput()` provided by `android.content.Context` library for opening and writing to a file). All occurrences of critical function calls are identified and labeled. We identified user triggers as a data input or action to a program as described in Section 2. Our framework analyzes Java bytecode or source code. It statically constructs the dependence graph that captures the data consumption relations and the Intent-specific control flow in Android apps to identify the directed paths between user triggers and entry points to methods providing critical system services.

Our static analysis framework is implemented in Java with 2,494 lines of source code. We convert Android app code (APK) from the dex format to a jar file using the *Dare* tool [41]. [2]

We evaluated 482 known Android malware samples collected by [54]. We also evaluated 708 popular free real-world Android apps collected in December 2012 from Google Play and alternative markets covering various categories. We assume that the trustworthiness of these free apps is unknown and they may be malware or contain malicious components. Table 2 shows a statistical summary of the Android apps used in our evaluation for both free popular apps and malicious apps. Our evaluation is to answer the following questions:

1. How many apps (free popular or malicious ones) conform to our user-intention dependence hypothesis?
2. Can our method find new malware from free popular apps?
3. What is the performance time of our analysis?

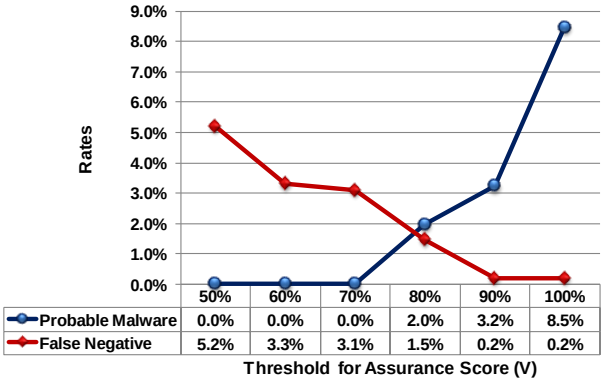
4.1 Analysis Accuracy

In addition to the assurance score V, we reported for each analyzed app the number of user triggers, the total number of sensitive API calls (operations), the list of the sensitive API calls used, and the number of operations with valid user triggers. Table 3 summarizes the assurance scores of the apps, namely, the percentage of critical API calls having the required dependence property.

Given a threshold $T \in (0, 100\%]$ for assurance scores, we analyze the assurance scores in two groups as follows.

- *The false negative* rate is computed for known malware samples. It is the percentage of apps whose assurance scores are higher than or equal to the threshold T. False negative rate represents undesirable detection misses.
- *The probable malware* rate is computed for free popular apps. It is the percentage of apps whose assurance scores are lower than the required threshold T. Because the trustworthiness of these apps is unknown, we further investigated each of these cases manually.

Detection results varied according to the assurance threshold chosen. We illustrate how false negative and probable malware rates change with thresholds in Figure 2, which is further explained next.



	50%	60%	70%	80%	90%	100%
Probable Malware	0.0%	0.0%	0.0%	2.0%	3.2%	8.5%
False Negative	5.2%	3.3%	3.1%	1.5%	0.2%	0.2%

Figure 2: Detection results w.r.t. various assurance thresholds. False negative rates (detection misses) are reported for known malware samples. For free popular apps, the percentage of apps that have-lower-than-the-threshold scores is reported. This percentage is referred to as the probable malware rate.

4.1.1 Known Malicous Apps

The analysis results of the known malicious apps are shown in Table 3 and Table 4. Most of them gave low assurance scores; they have a significant number of sensitive API calls without proper user triggers. In particular, 313 malware samples out of 482 gave 0% assurance values meaning that none of their critical function calls has user triggers. The majority of the rest of the 169 samples have lower than 50% assurance values. The only malware having 100% score is a phishing app, *FakeNetflix*. FakeNeflix malware provides a fake user interface to trick the user to enter her or his Netflix credential information. This app is a standalone phishing app which behaves similarly to a legitimate app in terms of using user

[2] We also tried the *dex2jar* tool [1], however, the conversion fails in many cases.

Table 2: Statistics of analysis for all apps analyzed.

	Free Popular Apps			Malicious Apps		
Statistic	Min	Max	Mean	Min	Max	Mean
Bytecode Size (KB)	20.13	869.84	178.84	4.75	1810.26	161.09
Class Count	7	1897	135.67	7	240	77.87
Method Count	8	9351	613.24	5	1202	338.61
Sensitive API Call Count	1	93	20.93	1	72	8.87
DDG Node Count	32	24792	7016.13	97	22885	7402.47

Table 3: Summary of evaluation results.

Assurance Score (V)	**# of Free Popular Apps**	**# of Malware**
0%	0	313
(0% - 10%)	0	68
[10% - 20%)	0	4
[20% - 30%)	0	20
[30% - 40%)	0	27
[40% - 50%)	0	25
[50% - 60%)	0	9
[60% - 70%)	0	1
[70% - 80%)	14	8
[80% - 90%)	9	6
[90% - 100%)	37	0
100%	648	1
Number of Samples	708	482

triggers to launch sensitive operations. This type of malware behavior can circumvent our approach and lead to false negatives. Detecting malicious social engineering apps is challenging. App certification and user education approaches are more effective than program analysis.

The Android malware samples that we analyzed perform malicious functionality, 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*) [3].

Operations in some malware samples (35.1%) have proper user triggers. Their assurance scores are non-zero. The reason is that these malware samples are not stand alone apps; they are bundled with other legitimate apps that require user interactions. *ADRD*, *DroidDream*, and *Geinimi* are examples of repackaged malware in benign apps.

Table 4 shows types of API calls used by a subset of the malware. Our manual review reveals that some examples of sensitive API calls in malware are related to writing and sending information through the network, sending unauthorized SMS messages, executing system processes, and accessing user's private data. For example, *Asroot* and *BaseBridge* use `Runtime.exec()` to execute system processes.

4.1.2 Free Popular Apps

The analysis results of the free popular apps are shown in Table 3 and Table 5. Most of the free popular apps conform to our hypothesis. For example, 98% of apps have assurance scores greater than or equal to 80% (i.e., for each of these apps 80% or more of their critical function calls have the required user-trigger dependences). In the most strict case where the threshold is set to 100%, the probable malware rate is 8.5% (i.e., 60 out of the 708 apps gave less than 100% assurance scores).

We manually inspected these 60 probable malware apps, to understand their behaviors. Table 5 depicts the analysis results for some of the 60 free popular apps that had lower than 100% assurance scores. The table shows the total number of sensitive API calls, assurance score value, and the name of sensitive API calls that are not triggered by the user intention. The problematic operations involve access to private information such as phone identifier and locations. These operations are exclusively in the ads and/or analytics libraries. Figure 3 shows the breakdown between non-policy-conforming uses in these apps.

Many of the advertisement or analytics libraries in these 60 apps are heavily obfuscated. We describe two such apps. For example, in application `com.androidscreenshotapptool.free` [4], we found API calls to `getLongitude()`, `getLatitude()`

[3]The malware naming convention follows [54].

[4]This app allows the user to create a screenshot of Android.

Table 4: A subset of known malicious apps analyzed.

Malware Name*	# of Sensitive API Calls	Assurance Score (V)	Distinct API Calls Without User Intention**
Asroot	15	73.3%	write(), read(), readLine(), Runtime.exec()
BaseBridge	74	63.5%	write(byte[]), read(), openFileInput(), Runtime.exec(), getLongitude(), getLatitude(), getDeviceId(), getNetworkOperatorName()
DroidKungFu1	60	63.3%	write(byte[]), read(), openFileInput(java.lang.String), Runtime.exec(), getLongitude(), getLatitude(), getDeviceId(), getNetworkOperatorName()
DroidKungFu2	18	5.6%	write(byte[]), read(), Runtime.exec(), getDeviceId(), getLine1Number()
GingerMaster	57	75.4%	write(byte[]), read(), java.io.File: boolean delete() Runtime.exec(), android.os.AsyncTask execute();
HippoSMS	9	33.3%	sendTextMessage(), write(), read(byte[]), openFileInput(), openFileOutput(), java.io.File: boolean delete()
NickySpy	17	58.8%	sendTextMessage(), getDeviceId(), read(), java.io.File: boolean delete()
Pjapps	40	47.5%	org.apache.http.client.HttpClient: java.lang.Object execute(), write(byte[]), readLine(), getCellLocation(), getLine1Number() getDeviceId(), getSimSerialNumber(), getSubscriberId(), getNetworkOperator()
Tapsnake	12	25%	org.apache.http.client.HttpClient: java.lang.Object execute(), getLongitude(), getLatitude(), getAccuracy(), openFileOutput(), openFileInput()
Walkinwat	7	14.3%	org.apache.http.client.HttpClient: java.lang.Object execute(), getLongitude(), getLatitude(), getDeviceId(), sendTextMessage()

*This is malware family name. Each malware sample belongs to a family has different analysis results.
**Some method signatures are omitted due to space limitation.

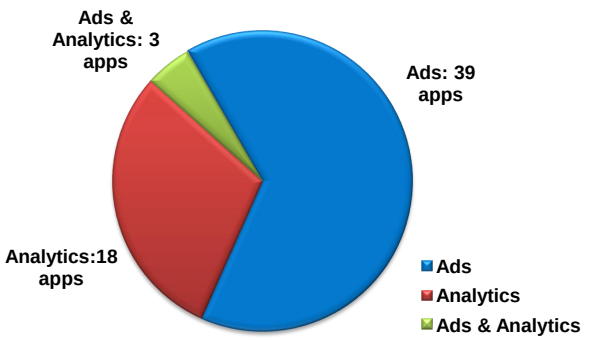


Figure 3: The breakdown of causes of free popular apps that lack proper trigger-operation dependences.

functions inside `com.flurry.android` analytics library without user triggers. In application `com.aol.mobile.aolapp` [5], it has calls to `getDeviceId()` functions inside `com.flurry.android` library without user triggers. Some of the libraries were mentioned in other work, e.g., Enck *et al.* [19] deduced that code obfuscation in advertisement or analytics libraries is a form of intellectual property protection. Still, little is known on the assurance of these libraries.

Location and device information may be sensitive. Our analysis pinpoints the exact places where the API calls are made to access these information without any user triggers. From the assurance assessment perspective, knowing the lack of proper user-based dependences in those sensitive function calls is useful. If certain ad or analytical libraries are known to be legitimate, then they may be put on whitelists to avoid triggering false alarms.

We checked all the collected free apps by using Virus-

[5]This app allows the user to read the top stories from AOL.

Table 5: A subset of free popular apps that lack proper trigger-operation dependences. The cause indicates the reason of lacking trigger-operation dependences.

App Name	# of Sensitive API Calls	Assurance Score (V)	Distinct API Calls Without User Intention*	Cause	Category
com.ab.labyNaruto	63	95.2%	read(byte[]), readLine()	ads	Libraries & Demo
com.androidscreenshotapptool.free	38	81.6%	write(byte[]), read(), readLine(), getAccuracy(), getLongitude(), getLatitude()	analytics	App Widgets
com.androidtrainer.survive	9	88.89%	readLine()	ads	Books & Ref
com.aol.mobile.aolapp	37	78.3%	write(byte[]), readLine(), getDeviceId()	analytics	News & Magazines
com.hd.peliculashd	39	94.9%	readLine()	ads	Entertainment
com.indeed.android.jobsearch	25	96%	readLine()	ads	Business
com.intellijoy.android.shapes	17	70.6%	readLine(), getAccuracy(), getLongitude(), getLatitude()	ads	Education
com.max.SurvivalGuide	11	90.9%	readLine()	ads	Books & Ref
com.pcvirt.livewallpaper.ocean	10	90%	readLine()	ads	Wallpaper
com.rechild.advancedtaskkiller	17	94.1%	readLine()	ads	App Widgets

*Some method signatures are omitted due to space limitation.

Total [4], which uses more than 40 different antivirus products and scan engines. None of them have known malware signatures. Signature-based scanning is unable to locate potential causes of unintended operations; neither is the default Android permission system. In comparison, our tool provides fine-grained reports on app operations and attempts to reason their causes and purposes.

4.2 Efficiency Evaluation

The experiments were conducted on a computer which has 3.0GHz Intel Core 2 Duo CPU E8400 processor and 3GB of RAM. We measured the execution time of our analysis, specifically steps *(iii)* and *(iv)* explained in Section 3. The average processing time for an app is about 249.48 seconds. This processing time does not include the time required to convert the dex format to jar.

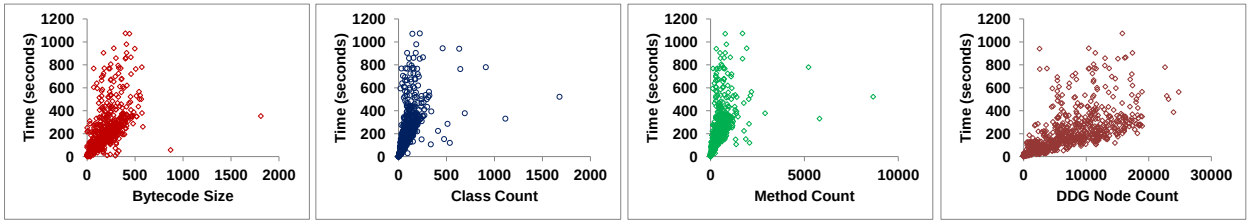
The execution time varies significantly across different apps. We studied the impact of four specific factors (bytecode size, method count, class count, and node count in the constructed graph) that may affect the execution time as shown in Figure 4. The execution time positively correlates with the size of the apps and the number of components therein. Because only a partial DDG is traversed for finding dependence paths, the increase in execution time with the total number of DDG nodes is relatively slow, compared to the increase in execution time with the number of classes and methods.

Summary The experimental results provide encouraging evidence supporting our hypothesis that critical operations in legitimate apps are mostly triggered by user inputs or actions. They demonstrate the feasibility of our static user-intention dependence analysis as a useful app classification method. When using 80% as the assurance threshold, our results gave a reasonable false negative rate (1.5%) for known malware and a probable malware rate (2%) for free popular apps studied [6]. The latter may require further inspection through manual effort and the use of other complementary analysis tools to understand their behaviors. Our future work will investigate more complex dependence requirements that may involve multiple operations or triggers to improve the classification accuracy.

5 Security Analysis

Our static analysis aims to identify possible unauthorized privilege operations which do not have dependences with user inputs or actions in the program. Our analysis method on trigger-operation dependences realizes this goal. Critical 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:

[6]Because of the lack of ground truth for free popular apps, we are unable to compute the false negative rate for that group.



(a) Bytecode Size vs. Time (b) Num. of Classes vs. Time (c) Num. of Methods vs. Time (d) Num. of DDG Node vs. Time

Figure 4: Performance time analysis.

- *authorization related:* executing critical operations without proper user triggers, such as sending unauthorized SMS messages, subscribing to premium-rate services automatically, or granting unauthorized root privilege to apps.
- *confidentiality related:* accessing sensitive data items without proper user triggers, such as recording and uploading the victims 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 function calls, thus the superfluous user inputs can be identified.

Sources of inaccuracy.

Overestimation of trigger-operation dependence may cause false negatives in the analysis report (i.e., failing to detect potentially malicious operations in the app). Overestimation of data dependence may be due to conditional branches that are unpredictable statically, and certain dependence paths only exist under specific conditions. This type of 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 an calendar event), or triggered by incoming network events. These run-time 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. Hybrid approaches based on both static and dynamic analyses are needed for complete dependence analysis that involves more than the user intention triggers. Its realization remains an interesting open problem.

These overestimation and underestimation issues can be addressed as explained above. However, there are several limitations of the static user-intention program analysis approach that require completely different technical approaches.

Limitations.

Social engineering apps may demonstrate proper trigger-operation dependences, because of the seemingly conforming dependence paths between user triggers and critical operations. Therefore, due to the intrinsic nature of our user-intention analysis, it is not suitable for detecting social engineering apps. Instead, app certification and user education are two effective solutions.

Our analysis does not examine the content of variables, only dependences of operations. Operations involving potentially malicious content may possess the necessary dependence property. For example, a (malicious) app may append a URL to any outgoing SMS message sent by the user, where the URL refers to a compromised website. The problem is due to the lack of understanding of operation semantics, rather than the accuracy of dependence analysis. Enforcing the intended semantics of programs in general-purpose automated program analysis is challenging. One program analysis approach to this issue is to statistically characterize content and operations to identify normal patterns (e.g., analyzing the

frequency of occurrences of string concatenation operations involving URLs in legitimate apps) and then detect anomalous patterns.

Our analysis is on Java bytecode, which does not apply to native libraries. Some applications utilize native libraries via Java Native Interface (JNI) to incorporate the C/C++ libraries into the application. We found that 58 out of 708 apps (8.19%) we studied include at least one native code library in their APK. Malicious apps can use obfuscation or Java reflection techniques to evade the standard static program analysis. The latter problem may be addressed by using dynamic taint analysis [40], [52] to provide insights about the program's runtime execution.

6 Related Work

We first explain the novelty of our work with respect to several closely related previous solutions, and then describe other notable results in the Android security and related literature.

Android-specific static analysis. Mapping the data flow or control flow dependences with static program analysis techniques allows one to understand expected behaviors of apps. Different security goals (or equivalently attack models) demand specialized analysis methods in Android. For example, SCanDroid [25] extracts security specifications from the app's manifest and checks whether data flows through the app are consistent with the stated specifications. Stowaway [21] aims to identify overprivileged Android apps by comparing the required and requested permissions based on mapping API calls used to permissions. CHEX [37] identifies potentially vulnerable component interfaces that are exposed to the public without proper access restrictions in Android apps. The authors utilized data-flow based reachability analysis. Definitions of sources, sinks, and policies are different from our work due to different security goals.

RiskRanker [30] aims to detect Android malware using *i)* a set of vulnerability specific-signatures and *ii)* control flow and intra-method data flow analysis looking for suspicious behavior signatures (e.g., calls accessing certain sensitive data in a code path that also contains decryption (for deobfuscation) and execution methods on their dependence paths should raise red flags). For example, RiskRanker searches for the code paths that may cost the user money without implying such user interaction (e.g., clicking onscreen button). That detection is a special case of our user-intention dependence analysis with the specific definition of the trigger/source and the operation/sink. In comparison, our analysis systematically generalizes this user-intention dependence approach and has broader scopes in terms of trigger and operation definitions. Because of our focus on def-use dependence analysis, our method does not require general control flow analysis, only event-specific control flow necessary for bridging data flow dependences.

The approaches in AndroidLeaks [26], SCANDAL [33], and PiOS [17] for iOS platform follow the line of the classic confidentiality information flow; (e.g., labeling sensitive data/sources and potentially risky sinks (typically network API calls) and reporting when there are data dependence paths between them). Our security goal is to identify operations unintended by a user, as opposed to confidentiality analysis. As a result, our definition of sources is anything related to user inputs or actions and may or may not be sensitive. The work in Woodpecker [28] differs from all above; it focuses on smartphone firmware security, specifically evaluating potential permission leaks through high privileged pre-loaded apps in smartphone firmware. We compare the above solutions with our user-intention dependence (UID) analysis in Table 1 in the introduction.

Roesner et al. [46] proposed user-driven access control (UDAC) gadgets for granting permissions, instead of adding them previously through manifests or system prompts. Although our work is similar with UDAC in recognizing the importance of user intention in security design, the technical approaches are fundamentally different. Our static program analysis can be fully automated, whereas UDAC being a runtime solution requires user participation.

Mobile app characterization. Several large-scale characterization efforts on benign Android apps or malware apps in terms of their data and resource access permissions have been reported recently, including [19, 22, 54]. For example, ComDroid [8] identifies security vulnerabilities caused by Android inter-app communication. AdRisk [29] systematically studies large number of popular ad libraries used in Android apps to evaluate the potential risks. Zhou et al. [53] characterizes the evolvement of families of Android malware using fuzzy hashes that identify repackaged apps. DNADroid [10] detects cloned Android apps in the markets by comparing similarities of program dependency graphs.

Android permission systems. Researchers have shown that the Android default permission systems are inadequate for data and device protection, and proposed extensions and mitigation solutions. For example, Felt et al. [23] introduced the permission re-delegation problem in inter-app communication of Android and pointed out that the permission re-delegation problem is a special case of the confused deputy problem. Cryptographic authentication techniques were presented to prevent these problems in QUIRE [15]. The Kirin [20] framework provides a lightweight certification of Android apps to block the installation of potential unsafe apps based on certain

undesirable permission combination. In Saint [42], the authors proposed install-time permission granting policies as well as runtime inter-application communication policies for improved Android security. AdDroid [43] and AdSplit [47] proposed different approaches to separate the privileges between the ad library and its host app to eliminate the permissions requests done by the host app on behalf of its ad library.

Runtime app monitoring. Solutions in the runtime app monitoring category complement a static app analysis approaches such as our work. They typically either *i)* enforce security policies or *ii)* detect abnormal execution patterns through behavioral signatures or learning/mining techniques. For example, Dixon et al. [16] and Liu et al. [34] monitor power consumption of a device for anomalies. Crowdroid [7] performs clustering algorithms on activity and behavior data of running apps to classify them as benign or malicious. DroidRanger [55] detects known malicious apps in Android Markets. It extracts behavioral signatures of known malware samples, and then detects new samples of known malware families using the behavioral signatures. TaintDroid [18] uses dynamic taint analysis to report potential privacy leaks at runtime. Rastogi et al. [45] proposes the AppsPlayground framework which performs dynamic analysis of Android apps for the purpose of detecting malicious activities and privacy leaks. Aurasium [50] repackages apps to add user-level sandbox and security policies so that the app's runtime behavior can be restricted. ParanoidAndroid [44] performs runtime analysis for malware detection on a remote server in the cloud. DroidScope [51] provides a virtualization environment for the purposes of dynamic analysis and information tracking of Android apps. AppFence [31] modifies Android OS to protect private data from being leaked by providing and imposing fine-grained privacy controls on existing apps. TISSA [56] proposes a privacy mode in Android platform which provides fine-grained control over user privacy.

Non-Android-specific static program analysis, malware characterization, and detection. Static program analysis has traditionally been applied to ensuring the data integrity and confidentiality of information flow, e.g., [13, 14, 35, 36, 39]. Using static program analysis for anomaly detection was first described by Wagner and Dean [49], then improved by [5, 6, 24, 27] with more accurate control flow analysis with calling context and call dependences.

Solutions such as dynamic system-wide taint analysis in Panorama [52] and malware behavior characterization (e.g., [9]) are inspirations to some of the aforementioned security solutions on Android. We omit their details and other notable related security solutions in non-Android-specific platforms due to space.

7 Conclusions and Future Work

We demonstrated the feasibility of user-intention-based static dependence analysis in assessing the assurance of programs, in particular Android apps. We explained the need for approximating user intentions in our static analysis. Our method computes the percentage of critical function calls that depend on some form of user inputs or actions through def-use analysis of the code. We call this percentage value an assurance score. Our approach can be applied to general user-centered programs and applications beyond the specific Android environment studied. Our experiments on 482 malicious apps and 708 free popular apps show high classification accuracy. These results suggest the promise of using our user-intention-based dependence analysis. Thus, our approach strengthens software assurance assessment which requires a comprehensive set of techniques.

For future work, we plan to generalize the dependence definitions to include non-user triggers, and also to utilize advanced program analysis techniques to further improve the accuracy. For the deployment perspective, we expect static app analysis tools to be used by cybersecurity and software experts for screening apps, as opposed to regular smartphone users. We will work on tools to provide informative and intuitive interpretation of the multiple dimensional analysis results from various tools to these experts and help them to further examine potential problematic code regions.

References

[1] dex2jar: A tool for converting Android's .dex format to Java's .class. https://code.google.com/p/dex2jar/.

[2] National information assurance glossary. Tech. rep. CNSS Instruction No. 4009 National Information Assurance Glossary.

[3] Soot: a Java optimization framework. http://www.sable.mcgill.ca/soot/.

[4] VirusTotal. https://www.virustotal.com/.

[5] ABADI, M., BUDIU, M., ÚLFAR ERLINGSSON, AND LIGATTI, J. Control-flow integrity: principles, implementations, and applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (2005), ACM, pp. 340–353.

[6] BHATKAR, S., CHATURVEDI, A., AND SEKAR, R. Dataflow anomaly detection. In *Proceedings of the IEEE Symposium on Security and Privacy* (2006), IEEE Computer Society, pp. 48–62.

[7] BURGUERA, I., ZURUTUZA, U., AND NADJM-TEHRANI, S. Crowdroid: behavior-based malware detection system for Android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM)* (2011), ACM, pp. 15–26.

[8] CHIN, E., FELT, A. P., GREENWOOD, K., AND WAGNER, D. Analyzing inter-application communication in Android. In *Proceedings of the 9th Int'l Conference on Mobile Systems, Applications, and Services* (2011), ACM, pp. 239–252.

[9] CHRISTODORESCU, M., JHA, S., AND KRUEGEL, C. Mining specifications of malicious behavior. In *Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering* (2007), ACM, pp. 5–14.

[10] CRUSSELL, J., GIBLER, C., AND CHEN, H. Attack of the clones: Detecting cloned applications on Android markets. In *Proceedings of the 17th European Symposium on Research in Computer Security (ESORICS)* (2012), vol. 7459 of *Lecture Notes in Computer Science*, Springer, pp. 37–54.

[11] CUI, W., KATZ, R. H., AND TIAN TAN, W. Binder: An extrusionbased break-in detector for personal computers. In *Proceedings of USENIX Annual Technical Conference* (2005), USENIX, pp. 363–366.

[12] DAVI, L., DMITRIENKO, A., SADEGHI, A.-R., AND WINANDY, M. Privilege escalation attacks on Android. In *Proceedings of the 13th International Conference on Information Security (ISC)* (2010), Springer-Verlag, pp. 346–360.

[13] DENNING, D. E. A lattice model of secure information flow. *Communication of ACM 19*, 5 (1976), 236–243.

[14] DENNING, D. E., AND DENNING, P. J. Certification of programs for secure information flow. *Communication of ACM 20*, 7 (1977), 504–513.

[15] DIETZ, M., SHEKHAR, S., PISETSKY, Y., SHU, A., AND WALLACH, D. S. QUIRE: Lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX conference on Security* (2011), USENIX Association.

[16] DIXON, B., JIANG, Y., JAIANTILAL, A., AND MISHRA, S. Location based power analysis to detect malicious code in smartphones. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* (2011), ACM, pp. 27–32.

[17] EGELE, M., KRUEGEL, C., KIRDA, E., AND VIGNA, G. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2011), The Internet Society.

[18] ENCK, W., GILBERT, P., GON CHUN, B., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation* (2010), USENIX Association, pp. 393–407.

[19] ENCK, W., OCTEAU, D., MCDANIEL, P., AND CHAUDHURI, S. A study of Android application security. In *Proceedings of the 20th USENIX conference on Security* (2011), USENIX Association.

[20] ENCK, W., ONGTANG, M., AND MCDANIEL, P. D. On lightweight mobile phone application certification. In *Proceedings of the ACM Conference on Computer and Communications Security* (2009), ACM, pp. 235–245.

[21] FELT, A. P., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (2011), ACM, pp. 627–638.

[22] FELT, A. P., FINIFTER, M., CHIN, E., HANNA, S., AND WAGNER, D. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM)* (2011), ACM, pp. 3–14.

[23] FELT, A. P., WANG, H. J., MOSHCHUK, A., HANNA, S., AND CHIN, E. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Security Symposium* (2011), USENIX Association.

[24] FENG, H., GIFFIN, J., HUANG, Y., JHA, S., LEE, W., AND MILLER, B. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of IEEE Symposium on Security and Privacy* (2004), IEEE Computer Society.

[25] FUCHS, A. P., CHAUDHURI, A., AND FOSTER, J. S. SCanDroid: Automated security certification of Android applications, 2009. Technical report, University of Maryland.

[26] GIBLER, C., CRUSSELL, J., ERICKSON, J., AND CHEN, H. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *Proceedings of the 5th International Conference on Trust & Trustworthy Computing (TRUST)* (2012), vol. 7344 of *Lecture Notes in Computer Science*, Springer, pp. 291–307.

[27] GIFFIN, J., JHA, S., AND MILLER, B. Efficient context-sensitive intrusion detection. In *Proceedings of the Network and Distributed System Security Symposium* (2004), The Internet Society.

[28] GRACE, M., ZHOU, Y., WANG, Z., AND JIANG, X. Systematic detection of capability leaks in stock Android smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)* (2012).

[29] GRACE, M. C., ZHOU, W., JIANG, X., AND SADEGHI, A.-R. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WISEC)* (2012), ACM, pp. 101–112.

[30] GRACE, M. C., ZHOU, Y., ZHANG, Q., ZOU, S., AND JIANG, X. RiskRanker: scalable and accurate zero-day Android malware detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)* (2012), ACM, pp. 281–294.

[31] HORNYACK, P., HAN, S., JUNG, J., SCHECHTER, S. E., AND WETHERALL, D. These aren't the droids you're looking for: retrofitting Android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)* (2011), ACM, pp. 639–652.

[32] HORWITZ, S., REPS, T., AND BINKLEY, D. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems 12* (1990), 26–60.

[33] KIM, J., YOON, Y., YI, K., AND SHIN, J. SCANDAL: Static analyzer for detecting privacy leaks in android applications. In *Proceedings of the Workshop on Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy* (2012).

[34] LIU, L., YAN, G., ZHANG, X., AND CHEN, S. VirusMeter: Preventing your cellphone from spies. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection* (2009), Springer, pp. 244–264.

[35] LIU, Y., AND MILANOVA, A. Practical static analysis for inference of security-related program properties. In *Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC)* (2009), IEEE Computer Society, pp. 50–59.

[36] LIU, Y., AND MILANOVA, A. Static information flow analysis with handling of implicit flows and a study on effects of implicit flows vs explicit flows. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR)* (2010), IEEE Computer Society, pp. 146–155.

[37] LU, L., LI, Z., WU, Z., LEE, W., AND JIANG, G. CHEX: statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2012), ACM, pp. 229–240.

[38] MERCEDES, K., AND WINOGRAD, T. Enhancing the development life cycle to produce secure software. Tech. rep., 2008. Data and Analysis Center for Software.

[39] MILANOVA, A., AND RYDER, B. G. Annotated inclusion constraints for precise flow analysis. In *Proceedings of the 21st IEEE International Conference on Software Maintenance(ICSM)* (2005), IEEE Computer Society, pp. 187–196.

[40] NEWSOME, J., AND SONG, D. X. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium* (2005), The Internet Society.

[41] OCTEAU, D., JHA, S., AND MCDANIEL, P. Retargeting Android applications to Java bytecode. In *Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)* (2012), ACM.

[42] ONGTANG, M., MCLAUGHLIN, S. E., ENCK, W., AND MCDANIEL, P. D. Semantically rich application-centric security in Android. In *Proceedings of the 25th Annual Computer Security Applications Conference* (2009), IEEE Computer Society, pp. 340–349.

[43] PEARCE, P., FELT, A. P., NUNEZ, G., AND WAGNER, D. AdDroid: privilege separation for applications and advertisers in Android. In *Proceedings of 7th ACM Symposium on Information, Compuer and Communications Security (ASIACCS)* (2012), ACM, pp. 71–72.

[44] PORTOKALIDIS, G., HOMBURG, P., ANAGNOSTAKIS, K., AND BOS, H. Paranoid Android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)* (2010), ACM, pp. 347–356.

[45] RASTOGI, V., CHEN, Y., AND ENCK, W. AppsPlayground: Automatic large-scale dynamic analysis of Android applications. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY)* (2013), ACM.

[46] ROESNER, F., KOHNO, T., MOSHCHUK, A., PARNO, B., WANG, H. J., AND COWAN, C. User-driven access control: Rethinking permission granting in modern operating systems. In *Proceedings of the IEEE Symposium on Security and Privacy* (2012), IEEE Computer Society, pp. 224–238.

[47] SHEKHAR, S., DIETZ, M., AND WALLACH, D. S. AdSplit: separating smartphone advertising from applications. In *Proceedings of the 21st USENIX conference on Security symposium* (2012), USENIX Association.

[48] SHIRLEY, J., AND EVANS, D. The user is not the enemy: Fighting malware by tracking user intentions. In *Proceedings of the Workshop on New Security Paradigms (NSPW)* (2008), ACM, pp. 33–45.

[49] WAGNER, D., AND DEAN, D. Intrusion detection via static analysis. In *Proceedings of IEEE Symposium on Security and Privacy* (2001), IEEE Computer Society, pp. 156–68.

[50] XU, R., SAÏDI, H., AND ANDERSON, R. Aurasium: practical policy enforcement for Android applications. In *Proceedings of the 21st USENIX conference on Security symposium* (2012), USENIX Association.

[51] YAN, L. K., AND YIN, H. DroidScope: seamlessly reconstructing the os and dalvik semantic views for dynamic Android malware analysis. In *Proceedings of the 21st USENIX conference on Security symposium* (2012), USENIX Association.

[52] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the ACM Conference on Computer and Communications Security* (2007), ACM, pp. 116–127.

[53] ZHOU, W., ZHOU, Y., JIANG, X., AND NING, P. DroidMOSS: Detecting repackaged smartphone applications in third-party Android marketplaces. In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy* (2012), ACM.

[54] ZHOU, Y., AND JIANG, X. Dissecting Android malware: Characterization and evolution. In *Proceedings of the IEEE Symposium on Security and Privacy* (2012), pp. 95–109.

[55] ZHOU, Y., WANG, Z., ZHOU, W., AND JIANG, X. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)* (2012).

[56] ZHOU, Y., ZHANG, X., JIANG, X., AND FREEH, V. W. Taming information-stealing smartphone applications (on Android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing (TRUST)* (2011), Springer, pp. 93–107.