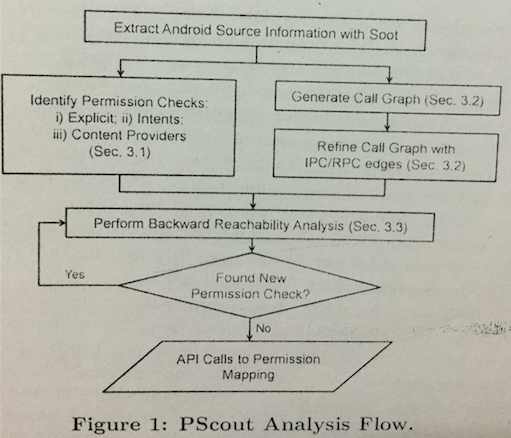
PScout: Analyzing the Android Permission Speciﬁcation

Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang and David Lie Dept. of Electrical and Computer Engineering University of Toronto, Canada

**Motivation:** smartphone OS uses the permission system to protect users’ privacy, it declares what sensitive resource the app will use, has the user agree with the this request when they install the app. But as the permission system becomes more common, questions has risen: the documentation of the permission system is incomplete, the required permissions of many undocumented APIs used by third party apps are not checked. The heavy interconnection of different API calls with different permissions. This paper introduces PScout to extract more complete permission specifications from android and map them with API calls (totally 17 thousand mappings).

**Idea:** To extract the permission specifications from the android framework, Pscout extract permission specification with Soot from the bytecode. It extracts three types of operations that will succeed or fail depending on the permission held by the app: (1) explicit call to checkpermission function; (2)methods involving intents; (3) methods involving content provider. It then builds a call graph over the entire android framework including IPCs and RPCs. Finally, it performs a backwards reachability traversal over the graph to identify all API calls that could reach a particular permission check.



This paper makes two summaries through the experiment: (1) There is little redundancy in the android permission specification. (2) The permission system is broad but not heavily interconnected. Permissions that have many API mappings tend to protect generic system resources rather than user data and fewer permission checks.

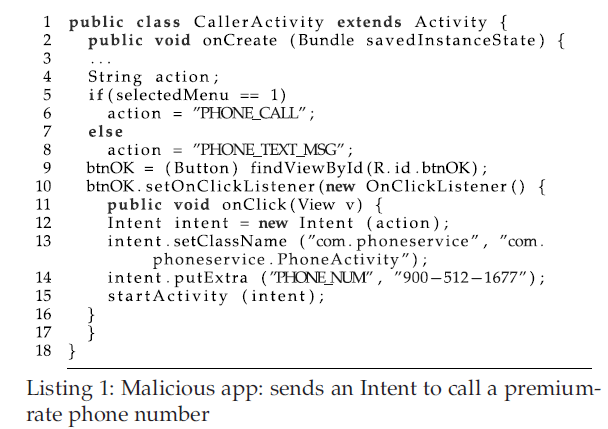
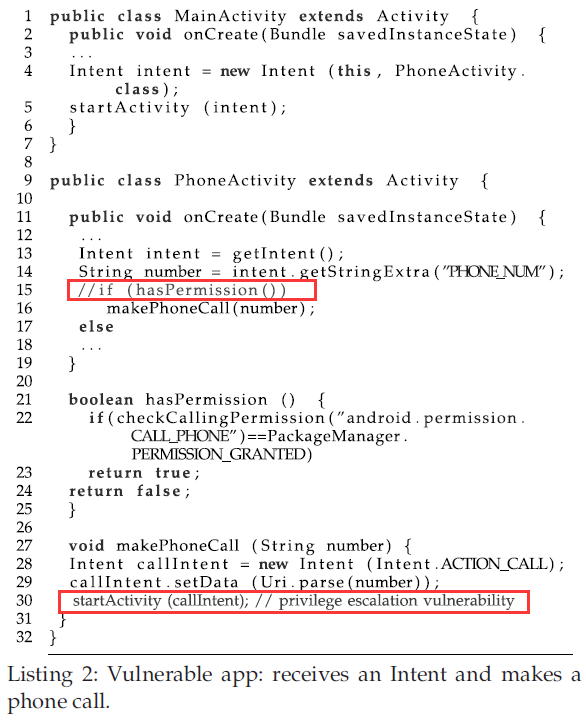
This paper mainly discusses about the permission specification and maps permissions to API calls, which is useful for other analysis. It has been applied in CONVERT in the following paper.

COVERT: Compositional Analysis of Android Inter-App Permission Leakage

Hamid Bagheri, Member, IEEE, Alireza Sadeghi, Joshua Garcia, and Sam Malek, Member, IEEE

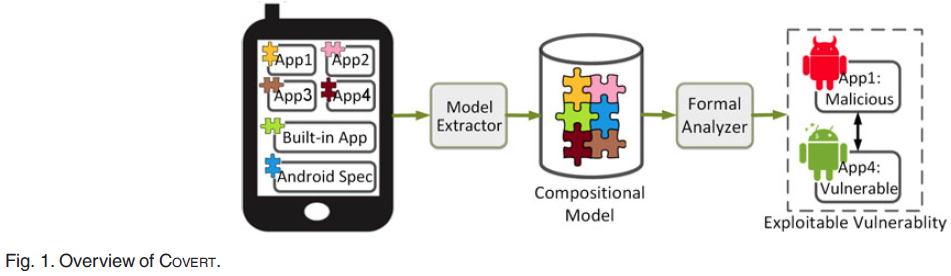
**Motivation:** Android, as the most popular platform for mobile devices, shares data and services among applications using an inter-app communication system. The access to resources is controlled by the android permission system. Currently, Android’s enforcement of permission is at the level of individual apps, allowing multiple malicious apps to collude and combine their permissions to attack the vulnerable apps, perform harmful actions which are beyond their individual privilege. Current approaches such as information leakage *[E. Chin et al.: Analyzing inter-application communication in android, 2011]* and least-privilege principles *[A. P. Felt et al.: Android permissions demystified, CCS 2011]* are designed to detect vuls (short for vulnerability) in a single app, but fail to detect vuls caused by the interaction of multiple apps. This paper present the compositional analysis of android inter-app vuls to tackle this problem.

**Motivating example:** Listing 1 shows CallerActivity belonging to a malicious app sending an Intent message to PhoneActivity (Listing 2) belonging to a vulnerable app for placing a call to a premium-rate telephone number. The vulnerability occurs on line 30 of Listing 2, where PhoneActivity initiates a system Intent of type ACTION\_CALL, resulting in a phone call.

The vulnerability occurs on line 30 of Listing 2. Although PhoneActivity has the makePhoneCall permission, it also needs to ensure that the sender of the original Intent message has the required permission to use the telephony service. The check is shown in hasPermission method of Listing 2, but in this particular example it does not get called. This kind of inter-app vuls is called privilege escalation.

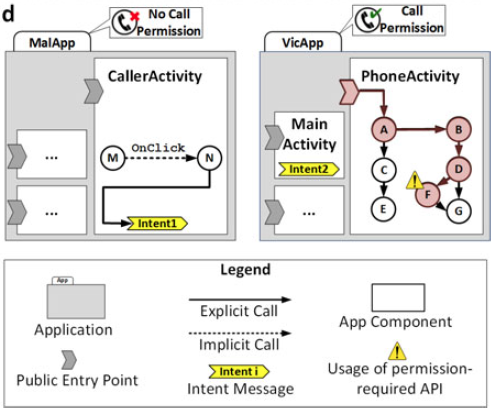
**Idea:** It combines static analysis with formal methods. It first decompiles several apks, statically analyzes the source code of each individual app to extracts relevant security specifications in a format suitable for formal verification. Then a formal analysis engine (e.g., model checker) is then used to verify whether it is safe for a combination of applications to be installed together.



* Model extractor:

The model is to determine the potential inter-process communication and reason about the security properties.

The extraction elements are: components, permissons that the app requires, public interface exposed by each app, intent creation and transmission, vul paths. The extracted model of Listing 1 and Listing 2 is as follows:



The activity transition graph can be constructed through intent. The vul path: An entry node is a node that activates a component. A vul node is an exported node (can be called by other apps or components) that request a certain permisson but without checking the caller permission. Entry 🡪destination path is considered to be a vul path. (The mapping relationships between permission and APIs are from PScout)

* Formal analyzer:

The approach automatically transforms the model into analyzable specification language, Alloy *[D. Jackson. Alloy: A lightweight object modelling notation, 2002]*, and verifies the model against certain properties. The properties that are expected to hold are extracted from the specification. It uses Alloy to define the android app fundamentals and constraints that every app must obey, and it is manually constructed once.

It develops assertions that model a set of security properties required to be checked. vulnerability assertions are manually constructed once and do not change, unless there are substantial changes in Android that resolve or modify the known types of interapp Vulnerabilities.

**Evaluation:**

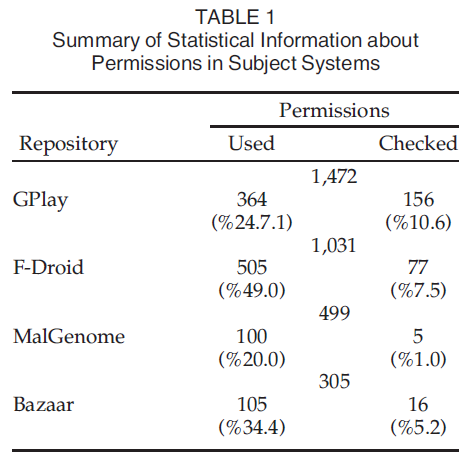


Table 1 outlines the amount of permissions requested by apps in each repository, along with the fraction of which is actually used through API calls, as well as enforced—depicted as checked—by the apps. The extraneous permissions that result in overprivilege are not susceptible to privilege escalation, unless they are actually used by the permission holders. On average, each app has about two unchecked but used permissions that could lead to exploitable vulnerabilities.

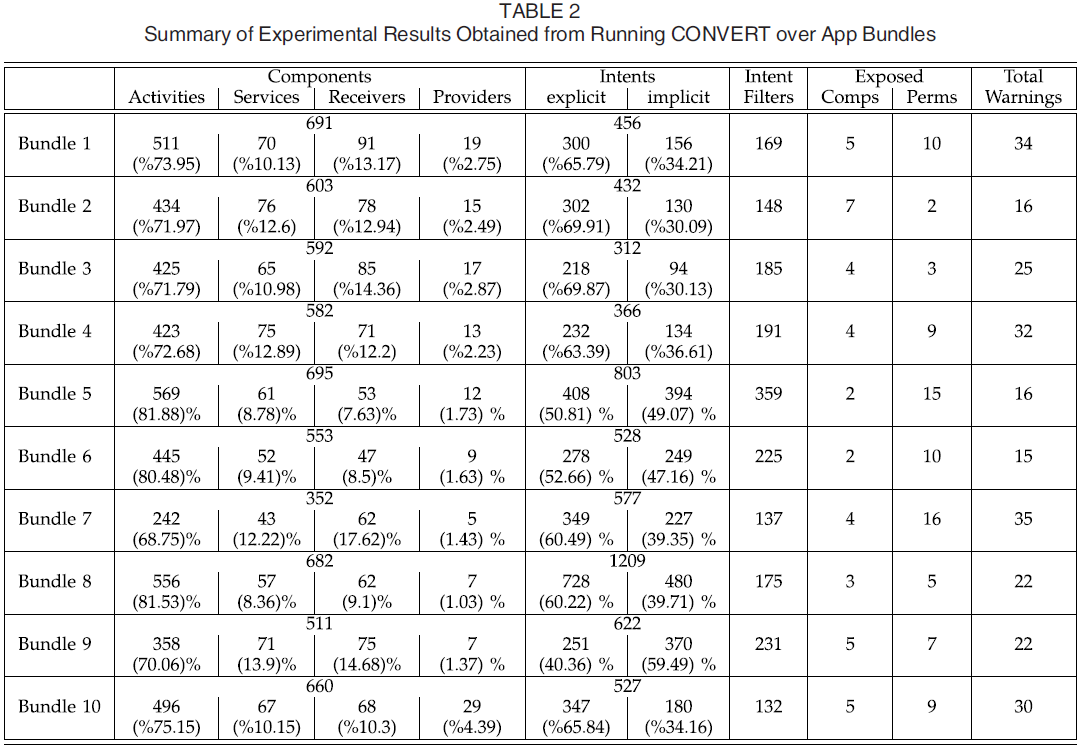


Table 2 summarizes the statistical results obtained through running COVERT on Android app bundles. Note that reported warnings are about potential security issues.

**Conclusion:** This paper employs static analysis to automatically recover models that reflect Android apps and interactions among them.