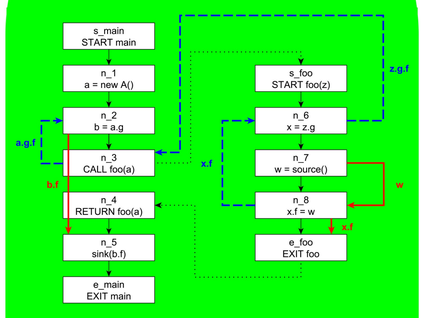
Scalable Information Flow Analysis for Android Apps

Shyam S, Poonam Kashyap, Aditya Kanade
Department of Computer Science and Automation

INTRODUCTION

The Android platform, since its release in2009, has prolifertated into the lives of a vast majority of people in multitude of ways. The Android environment is set up in such a way that many developers without professional training can publish and monetize their apps. This has led to a lot of security concerns regarding the accidental leakage of private information to unreliable entities. The permission system in Android can only prevent an app from accessing private information but gives no control over what is done with it. Since an app may legitimately require the use of private information for providing the user with the experience it has been developed for, just declining the permission is not a valid security measure in all cases. One chief way to detect the presence of such leaks is to perform static information flow analysis. The aim of such an analysis is to track the flow of private data without actual execution of the app, and to determine if this data could possibly be leaked. Although such an approach ensures that all possible paths in the app are covered, the time it takes is still quite high. This could question the feasibility of such analyses in an entirely automated testing scenario such as the security checks that the Google Play Store does before publishing apps. We propose to overcome this by introducing a hybrid approach. Our tool executes the app for a limited time and collects data from the execution which it then feeds into the static analysis thereby helping to complete it quicker.

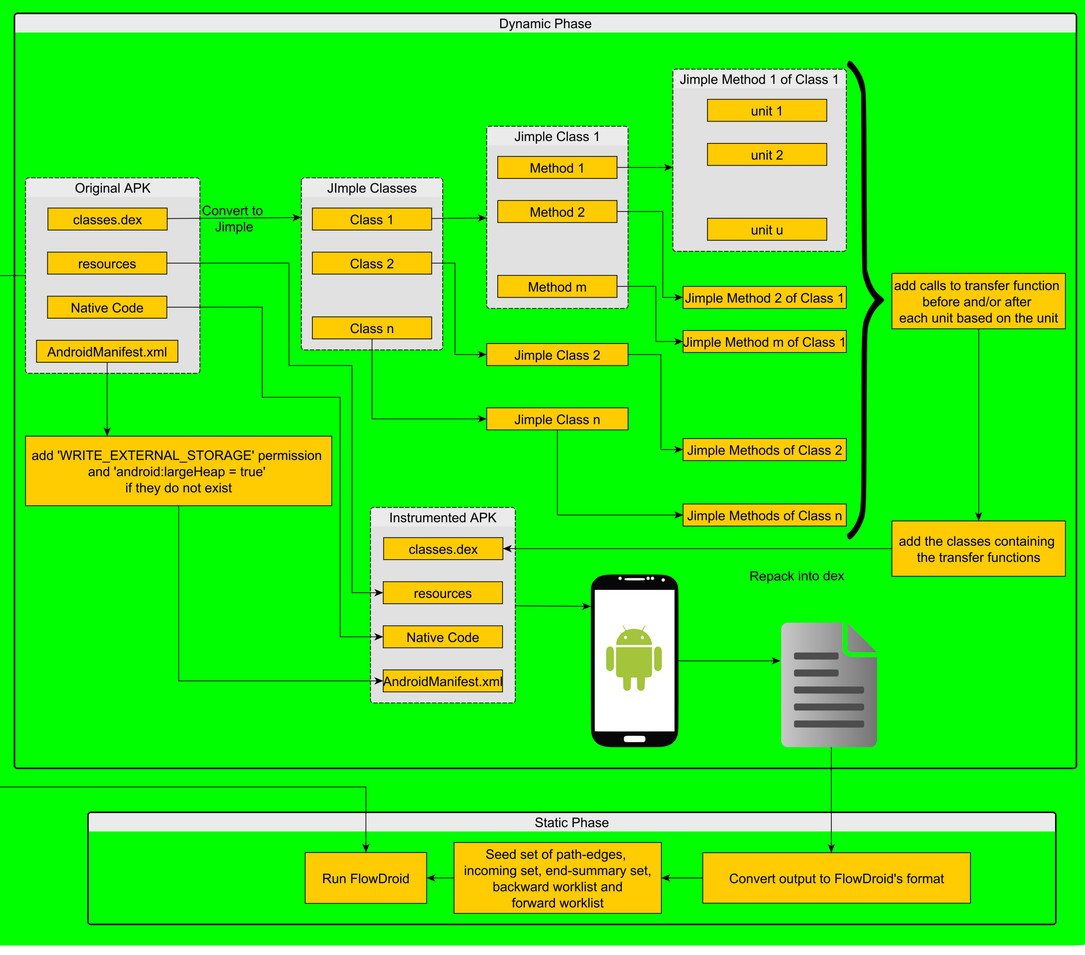
FLOWDROID EXAMPLE



1) At node n_7, 'w' gets tainted and is passed to node n_8.
2) Node n_8 taints x.f and passes it down to the next node.
3) Meanwhile, the analysis searches for aliases for x.f and traverses backward in the graph finding z.g.f at node n_6.
4) This then identifies a.g.f as an alias at node n_3 which then aliases with b.f at node n_2.
5) The taint on b.f then triggers a report at node n_5 when it flows into a sink, thus identifying a leak in the code.

PROPOSED HYBRID APPROACH

The hybrid approach first involves instrumenting the app to insert calls to our transfer functions, which manage the internal data structure. The data structure maintains information for each variable in a method with regard to which variables tainted it in the method scope. The transfer function is determined based on the type of statement being processed. Once the calls are inserted, the app is then remade by taking the resources and native code present in the original app. The transfer functions are designed to print out the tainted variables and the source of the taint(s) after each execution. This file will serve as the input to the static analysis. This requires us to modify the AndroidManifest.xml file to ask for the WRITE_EXTERNAL_STORAGE permission. We also request Android to provide us with an enlarged heap space to make room for the extra memory used in maintaining the data structure. Once the app is instrumented, it is then dynamically tested. This results in the transfer functions firing and generating the output required to seed the static analysis. The output also contains specific information to the static analysis to maintain soundness. This output is then parsed by our Seeder, which then seeds it into the static analysis. Below is the overall workflow showing this process in detail.



CHALLENGES

- Data structure to hold the information in memory during the dynamic run of the app without increasing the memory overhead of the app significantly.
- Design transfer functions, which do not over-approximate more than the transfer functions used by the static analysis.
- Maintain soundness in the presence of branches leading to paths not being explored in the dynamic run.
- Target specific methods to reduce processing overhead during the dynamic run of an app while ensuring sufficient coverage of methods which contain tainted data.
- Seed the obtained data such that there is no recomputation of data by the static analysis.