Android Malware Analysis: Leveraging Iterative Hybrid Analysis to Avoid Anti-Analysis Techniques

Prahathess Rengasamy, Joey Allen

Outline

Motivation
Goals
Reflection
Design
Implementation Details
Demo
Evaluation
Conclusion

Motivation

Motivation

01 Current hybrid analysis tools use static analysis to identify events to trigger using static analysis and passes them to dynamic analysis module to trigger those malicious events.

02 Several behaviors go undetected. For example, if a malicious developer leverages Java's reflection Library to obfuscate sensitive API calls, the tools would fail to identify this sensitive behavior ever occurred

03 we overcome the limitations by leveraging feedback loops to enhance the natural synergy between static and dynamic analysis

Goals

Goals

Find reflection call targets (the type of the objects and methods being reflectively created and invoked) through instrumenting and dynamically updating the analysis parameters and trigger them at runtime during dynamic analysis

Reflection

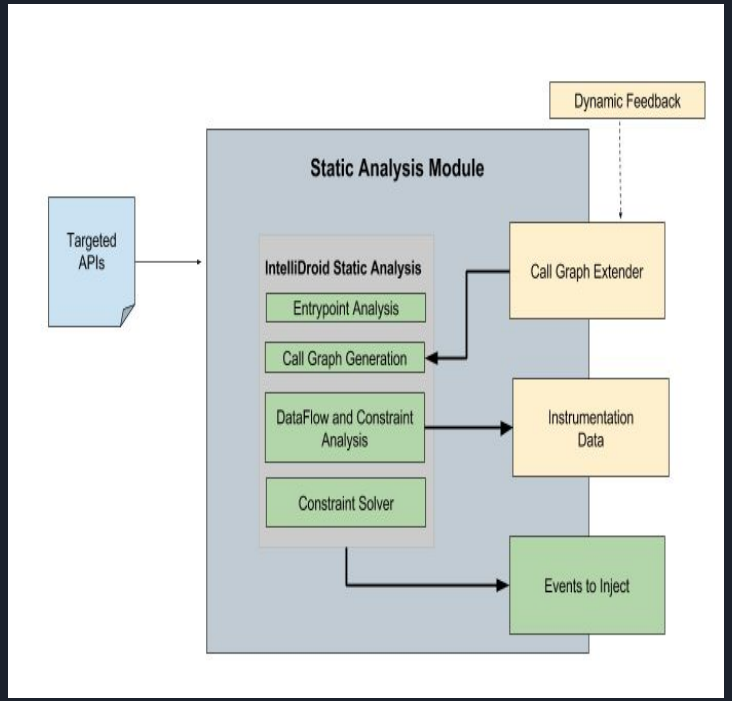
Reflection

- A dynamic programming language feature
- Enable developers to create objects and invoke methods by their names
- Used for
  - providing plugin and external library support
  - Invoke hidden APIs
- In addition to invoking them it is capable of :
  - Obtaining Method Objects
  - Method Parameters and Return Types
  - Invoking Methods using Method Object

Design

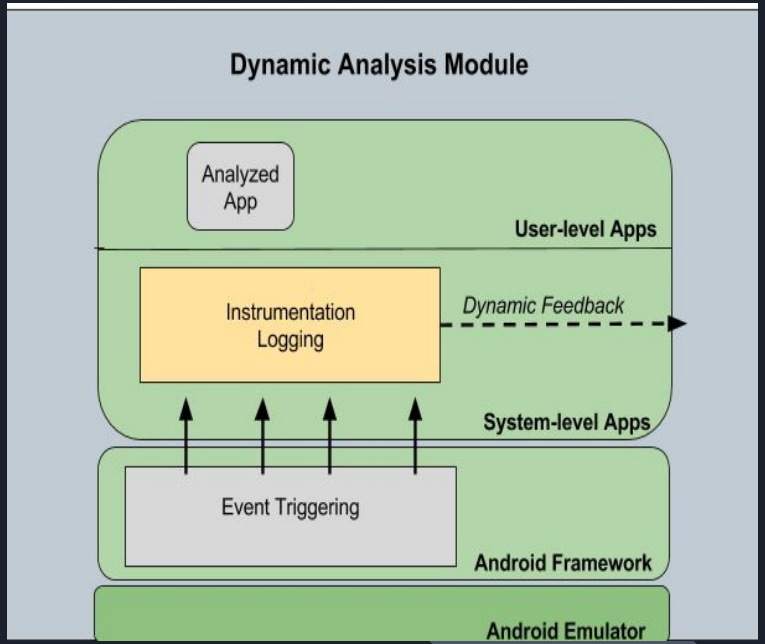
Design-Static Analysis Module

- The static analysis module in our architecture will enhance the state-of-the-art in two ways.
  - Instrumentation Info extension
  - second extension will be a Call Graph Extender
- Instrumentation Info extension
  - Instruments function calls and instructions
  - Resolve reflection calls and parameters
- Call Graph Extender
  - Use the resolved reflection information to extend the initial call-graph
  - With the extended call-graph the static module has more paths to analyze and resolves the application behavior better



Design-Dynamic Analysis Module

- Starting in Android 5.0, the Dalvik Virtual Machine (DVM) was depreciated and replaced with Android's new runtime environment, ART
- But still Dalvik bytecode is used to as an intermediate format and can be used to instrument apps.
- Instrumentation info module used to provide logging, such as the input parameters for Java Reflection Methods during runtime.
- The logged information will be then passed back to the static analysis module to allow for deeper analysis.



Implementation Details

Implementation Details-Instrumentation Module

- Uses soot to Instrument the Android application APK
- Uses a interesting methods file to get API/function calls to instrument
- This does the following changes to the original function body.
  - Print class name, method name, permission
  - If argument is primitive type, log Argument value, else Argument type
  - If argument is non-primitive, call toString method, print toString()
  - If return value is not void, follow the similar procedure to insert expression after a unit.
- Insert the required logic before each API call invocation and after each API call invocation.
- Since the application has been rewritten it signs the application using jarsigner
  - jarsigner -verbose -sigalg SHA1withDSA -digestalg SHA1 -keystore ../own-app/my-release-key.keystore $APK mykey -storepass 12341234

Implementation Details-Instrumentation Module

- Entrypoint methods Instrumented to log the parameters to resolve reflection calls
- Most significant reflective APIs that affect the static analysis are handled
  - Entry methods: Class.forName()
  - Member-instrospecting Methods: Method.getMethod(), Method.getDeclaredMethod(), Method.getMethods(), and Method.getDeclaredMethods()
  - Side-effect methods: Class.newInstance(), Method.invoke()

Implementation Details-Call Graph Module

- Uses wala and z3 constraint solver
- WALA provides support for basic static analysis, such as call graph generation, data flow analysis, alias analysis, and an intermediate representation based on SSA.
- A traversal for each event handler is then performed to identify the event handlers that may lead to a suspicious API invocation as defined by the user-specified suspicious behaviors. For each suspicious behavior that is found, a suspicious call path is extracted, which contains the sequence of method invocations from the event handler entry-point to the invocation of the suspicious API.
- The reflection calls resolved are read from the config file and they are treated as suspicious calls .
- For every reflection call a targeted path analysis is done which does the following:
    - Finds the class that declares the method implements a framework interface or extends a framework class, and the method itself must override a parent method in this interface or class;
    - Checks if the method is not called within the application code
    - Checks if the declaring class is instantiated at least once in the application code
    - A call graph is then generated using the event handlers as starting points for the code traversal

Demo

Evaluation

App Type	Total Reflection calls	Resolved Reflection calls	Call-Graph Nodes	Triggered Reflection calls
SingleView-Basic function calling	5	5	2710	5
SingleView-Internal API Calls	7	7	2885	7
SingleView-Content Provider	12	12	2974	12
SingleView-Broadcast Receivers	15	15	3265	15
Multiple Views-SMS-Messenger	12	12	3117	12
Multiple Views-Passing values via intents	10	10	2918	10
Multiple Views-Data Export from a content provider(Contacts)	16	16	3273	16
Multiple Views-Data Export from a content provider-UI(Contacts)	16	16	3534	16
Multiple Views -REST API calls	20	20	3860	20
Background Services-Socket Events	21	11	4034	11

Evaluation-Droid Bench

App Type	Reflection resolution	Triggered events
Reflection 1- Sensitive data is stored in a field of a reflective class and directly read out again and leaked	Full	2
Reflection 2- Sensitive data is stored in a field of a reflective class, read out again using a method implemented in the "unknown" class and leaked.	Full	3
Reflection 3- Sensitive data is stored using a setter in a reflective class, read back using a getter and then leaked. No type information on the target class is used.	Full	3
Reflection 4- Sensitive data is read using a function in a reflective class and leaked using another function in the same reflective class.	Full	4

Conclusion

Reflection analysis for Android apps for discovering the behaviors of reflective calls was added as a feature on top of Intellidroid.We advance the state-of-the-art reflection analysis for Android apps, by:

(1)Resolving reflection parameters

(2)Extending the call-graph to accommodate reflection methods

(3) demonstrating that resolved parameters can be used to generate constraints and triggered during runtime

Future Work

1. A feature extraction module integrated into the instrumentation steps and call graph steps will be a good add on
2. Stand alone analysis of background services will be useful direction to pursue
3. A machine learning component that gives a probability of maliciousness of an app based on the data gathered from the tool is a good addition.

Thank You

Q & A