Android Collusion Detection Using Linear Temporal Logic

Richard Allen and Markus Roggenbach[0000−0002−3819−2787]

Swansea University, Swansea, UK
{998184, M.Roggenbach}@swansea.ac.uk

The Android operating system has found many applications in devices such as phones, tablets, watches and smart TVs. It is characteristic for these that their primary use is not computation, though they include a computing system as an integral component. Furthermore, communication with other external computing systems is essential for their functionality, i.e., they are open systems by design. As such, they are vulnerable to typical cyber attacks, including information theft, service abuse, ransom-ware. Attacks happen to such extent that the McAfee Q1 2020 threat reports opens with the headline "Mobile Malware Is Playing Hide and Steal". [1]

What an app is allowed or is not allowed to do, Android regulates *statically* with a permission system. However, this system has known deficiencies. First, apps tend to request for excessive permissions. Second, users tend to grant permissions to apps without fully understanding the permissions or their implications in terms of risk. Third, the permissions system is concerned only with limiting the actions of individual apps. It is possible for two or more apps to collude for a malicious action by combining their permissions, even though each of the colluding apps is properly restricted by permissions. In a nutshell: under Android, the user does not know what data an app collects, what an app is doing with that data, or even who it shares it with: on the device or off the device.

In this context we suggest to *dynamically* monitor operating system calls on the device with techniques from runtime verification in order to alert the user of an Android device if there is a potential security breach. This will allow the user to react, e.g., by closing or removing the involved apps. Our approach comes with a number of advantages:

- The user is informed immediately when the device is under attack.
- It is independent of the application code.
- It is extendible to further security properties.

Our approach is similar to work by Bauer et al. [2]. However, we are also considering security properties that concern several different apps. Furthermore, we can make the guarantee that our monitor does not deteriorate over time, i.e., becomes slower with the number more observations.

In order to illustrate our approach, we use a simple form of information theft through app collusion as our running example [1]: two apps A and B combine their permissions in order to steal some information. To this end, app A first reads the information, then shares is via Android inter app communication with app B, which finally sends the information off the device. A typical example for A could be a contact managing app with read and write access for contacts, however, no Internet access; B could be a weather app, without access to contacts, however, access to the Internet in order to obtain the weather forecast. There is evidence of collusion based cyber attacks in the wild [3].

Android apps run in a 'sandbox'. Any access to a system resource requires a call to the operation system. Thus, *conceptually*, information theft through app collusion can be detected by monitoring operation system calls. For information theft through app collusion to possibly happen, the following sequence of calls is necessary: app A makes a query q to some resource protected by Android and then initiates an inter app communication on the device by a send call s; app B makes a receive call r and performs a publish call p. [2] It it possible to express such trace properties in linear temporal logic (LTL). For instance, the LTL formula

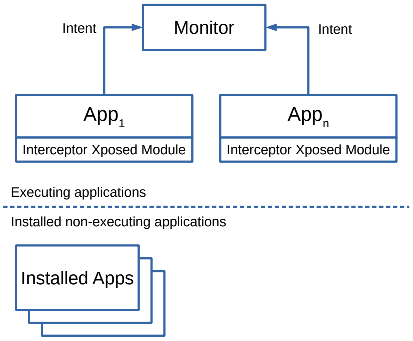
$$(\Box s \wedge (\neg s \; U \; \Box q)) \wedge (\Box r \wedge (\neg r \; U \; \Box s)) \wedge (\Box p \wedge (\neg p \; U \; \Box r))$$

[1] https://www.mcafee.com/content/dam/consumer/en-us/docs/2020-Mobile-Threat-Report.pdf

[2] Note that a sequence $\langle q, s, r, p \rangle$ of operation system calls describes information theft only, if the published data can be 'traced back' to the queried data.

is true if a trace includes the call pattern necessary for information theft through app collusion. It is possible to express other relevant security properties in LTL, "alert whenever an app takes an image from the camera" or "the microphone is never activated before a telephone call has been initiated or accepted".

In the *technical realisation* of our approach, our system architecture makes use of the Xposed framework [5] in order to intercept calls to security sensitive Android O/S functions. It adds a monitor app to the user's device which logs these calls and, at arrival of a new event, checks for each security property of interest if it is fulfilled or not.


Intent
Monitor
Intent
App$_1$
Interceptor Xposed Module
App$_n$
Interceptor Xposed Module
Executing applications
Installed non-executing applications
Installed Apps

In a first attempt to implement the monitor app, we utilised an algorithm by Rosu and Havelund [4] which appeared to be interesting thanks to its low complexity: the cost for evaluating a property φ on a trace t is $O(|t| * |\varphi|)$, i.e., is linear. However, as this algorithm traverses the trace t from the last event to the first event, it is impossible to re-use intermediate results from evaluating φ on t for evaluating φ on $t \frown \langle e \rangle$, i.e., the trace t extended by an event e. In our experiments this limits the trace length to about 60 events, beyond that, the device did not respond any more. Such length is far too short to be of any use in a real world scenario. Thus, we implemented a 'reverse' version of this algorithm that allows for re-use. Our new algorithm performs well. Traces can grow to arbitrary size (we have tested the algorithm up to a trace length of 100,000 events) without encountering any issues and evaluation time for new events is in the area of a single millisecond for the formula stated above. The price for this high performance is that our new algorithm does not support the LTL next operator.

By utilising colluding apps developed for testing purposes, cf. [1], we have successfully demonstrated that our monitoring approach effectively works on Android hardware and does not cause any significant performance reductions (occurrence of an event causes CPU usage in the area of 1.5%). However, further experimentation, in particular with monitoring simultaneously for different security properties, will be necessary to completely evaluate the performance properties of our approach. First experiments suggest a linear increase in CPU usage with the number of formulae monitored for.

References

1. M. Asavoae, J. Blasco, T. M. Chen, H. Kalutarage, I. Muttik, H. N. Nguyen, M. Roggenbach, and S. A. Shaikh. Detecting malicious collusion between mobile software applications: The Android case. In *Data Analytics and Decision Support for Cybersecurity*. Springer, 2017.
2. A. Bauer, J.-C. Küster, and G. Vegliach. Runtime verification meets Android security. In *NASA Formal Methods Symposium*, pages 174–180. Springer, 2012.
3. J. Blasco, T. Chen, I. Muttik, and M. Roggenbach. Detection of app collusion potential using logic programming. *Journal of Network and Computer Applications*, 105, 06 2017.
4. G. Rosu and K. Havelund. Synthesizing dynamic programming algorithms from linear temporal logic formulae. Technical report, 2001.
5. rovo89. Xposed framework. https://www.xda-developers.com/xposed-framework-hub/, 2021 (accessed March 15, 2021).