Name it?

Nataniel P. Borges Jr.
CISPA
Saarland University
D-66123 Saarbrücken, Germany
Email: nataniel.borges@cispa.saarland

Andreas Zeller
CISPA
Saarland University
D-66123 Saarbrücken, Germany
Email: andreas.zeller@uni-saarland.de

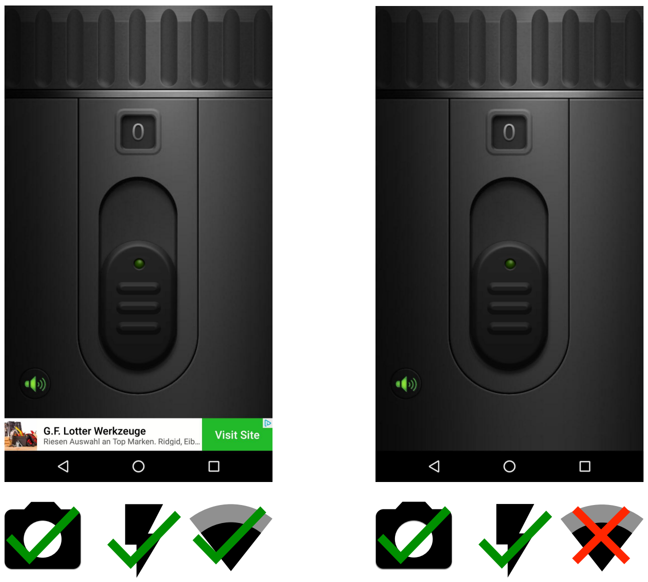
Abstract*—*[Nataniel: Pending]

I. Introduction

Android is currently the most popular software stack for mobile devices. It provides an API to access system resources, including camera, GPS and microphone. Third party developers use this API to create their own applications (apps) which are submitted to the Google Play Store, where they can be downloaded and installed by users [1]. Millions of users store privacy-sensitive information, such as contacts and photos, on mobile devices. Privacy-related studies [2] raise significant concerns regarding how apps handle sensitive data. While most apps do not maliciously mishandle sensitive data, problems occur due to faulty or sloppy implementations, as well as the use of third party libraries for monetization purposes, such as advertisements, whose internal behavior is frequently unknown for app developers.

A simplistic solution to prevent applications from leaking sensitive information would be to forbid apps from accessing any sensitive resource. This approach will, however, render several apps unusable. ~~For example:~~ photo apps without camera access or messaging apps without access to contacts.

While it is reasonable to assume that most apps have valid reasons to access privacy-sensitive resources, many apps are over privileged [3] and, even those apps where all permissions are actually used, frequently request access to more sensitive information than is necessary to perform their primary functionality. This excessive demand for sensitive information lead to such questions as: *why does my flashlight app need Internet access* [4]? To illustrate this question, Figure 1 displays two screenshots of the Surpax LED Flashlight app.



(a) All resources accessible (b) No Internet access

Fig. 1. Example of the LED Flashlight app with and Internet access

This flashlight app requires, among other resources, access to Camera, Flashlight and Internet. On Figure 1a, the screenshot displays the app when it has access to all requested sensitive resources, while on Figure 1b, the device has no Internet connection. On this example, without Internet access, the only visible unavailable functionality is the advertisement container.

The ~~OS~~ natively allows some sensitive information to be restricted with its runtime permission mechanism, however, this approach is limited in respect to usability as well as privacy management. Concerning usability, the user is not notified regarding the impact which will be caused by not granting access to a permission, which does not allow him to make informed decisions about allowing or denying access to permissions. Regarding privacy, the coarse granularity does not allow sensitive resources to be adequately adjusted – for instance, by allowing ~~*Camera*~~ access to enable the app to turn the flashlight on, involuntarily, the user grants access for the app to record videos or take pictures.

~~For a flashlight app, to record videos or to take pictures would characterize a misbehavior, that is, an action which is not expected by the user.~~ Several researches [5], [6], [7] have attempted to identify misbehaviors in Android apps. However, since apps frequently perform multiple tasks, the automatic identification of legitimate and malicious (or defective) functionality within an app is a difficult task. It requires the understanding of ~~*what the user expects the app to do* (intention) to be matched with *what the functionality actually does* (behavior)~~.

In this work we propose a different approach to mitigate the leakage of sensitive data caused by apps. Instead of attempting to identify misbehaviors in an app, we propose to measure the impact of restricting a specific API in a specific functionality, allowing users to define privacy policies according to their requirements. We apply an automated test input generation to

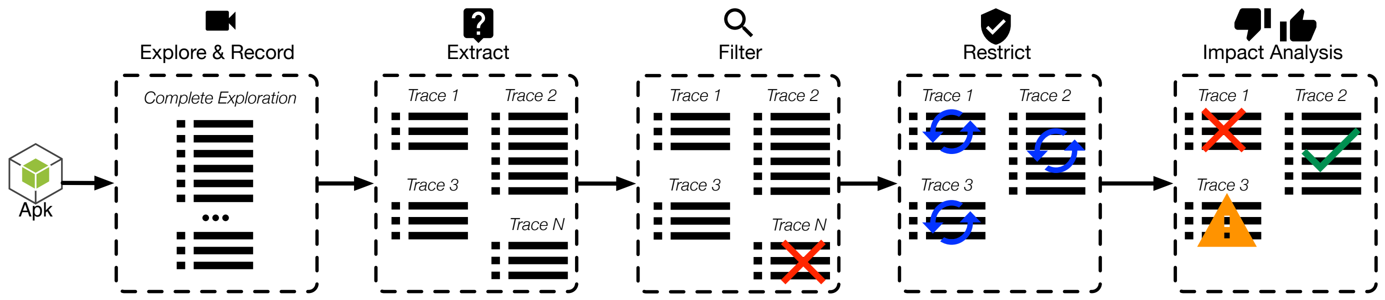


Fig. 2. Approach overview

explore applications for functionality and record its execution (exploration). ~~We then identify UI elements which access sensitive resources and attempt to reproduce the exploration while blocking different resources for different UI elements in order to measure the impact of restricting access to the resource.~~ This impact analysis can later be exploited to create privacy policies or assist users to make ~~informed decisions~~ about which sensitive resources the app should be able ot access.

In the remainder of this paper, after presenting a brief overview of the approach (Section II) and of Android UI (Section III), we introduce an approach to measure the impact of restricting access to API in specific functionality (Section IV). ~~Our experiments in Section VI demonstrate the suitability of the approach~~. After discussing related work (Section VIII), Section IX concludes the paper and outlines future work.

II. Approach Overview

Our approach to measure the impact of restricting access to an API in a specific functionality can be split into ~~5 steps: *Explore & Record*, *Analyze Filter* and *Restrict* and *Impact Analysis*~~ as illustrated in Figure 2.

During *Explore & Record*, an app is automatically explored using a test input generation tool and each interaction step – app state and chosen action – is recorded. Once this exploration finishes, the *Extract* step identify ~~segments~~ of the initial exploration in which sensitive resources are accessed. In the *Filter* step, the non reproducible segments are removed. We then restrict access to a sensitive resource while replaying the identified segments in the *Restrict*. Finally, during *Impact Analysis*, we measure how much the sensitive resource restriction affected the trace by comparing it to the initial one.

III. Background: Android UI

Android's user interface is hierarchically constituted of graphical (inheriting from *View*) and structuring (instance of *ViewGroup*) elements, which are ~~commonly referred to as *widgets*~~. The interaction between an app and the world occurs through ~~events~~. *System events*, such as "incoming call" or "message received", are generated by the OS and *user interaction events* are produced by user actions. Examples of user interactions are clicks, long clicks and swipes.

~~To automatically generate~~ test inputs for an app is equivalent to create and trigger a sequence of valid events on it. Therefore, to record and replay a complete exploration can be seen as to generate, store and re-execute a sequence of valid events.

In order to send an event to a specific widget, it is necessary to uniquely identify it on the active UI. While ~~internally~~ each widget possess a unique identifier, which allows the OS to ~~correctly~~ identify them~~. The~~ tools natively provided by Android to capture the current device's UI state hide this information. Instead, they provide only which widgets exist, how they are structured (parent-children relationship), alongside their current state (visible, enabled), content (textual and graphical) and display order. Based on these limitations, we uniquely define a widget as a ~~tuple of~~ (class type, resource ID, text, content description and location).

IV. Proposed Approach

We propose a combination between *automated test input generation*, *record-replay* and *policy enforcement* approaches to measure the impact of restricting API access on app functionality. Our approach is divided in 5 steps, which are described as follow.

A. Explore & Record

The goal of this step is to generate a baseline for the analysis by automatically recording the exploration of an application under test (AuT) by an automated test input generator. The test generator interacts with the app by interacting with widgets which are currently active in the AuT's UI, similarly to a human user. To interact with the widgets it simulates a human user, that is, it identifies the visible widgets which are currently actionable and sends an action to trigger the same event which a human would do by interacting with the device screen.

While exploring the AuT we *record* interactions I between the test generator and the AuT into an exploration log L. We represent each interaction as a triple $I = (U, A, P)$, where U is the *UI state* before the action, a is the *chosen action* and P is the set of *relevant Android APIs* invoked when executing A. Figure 3a illustrates a recorded exploration log with 11 interactions between the test generator and the AuT.

We model a state U as a set of widgets ($w \in U$), according to the definition from Section III, which allows our approach to be used in stock OS versions. We model an action A as a pair (t, w'), where t is the type of interaction between the test generator and the AuT and w' is the widget which

$$\begin{aligned}
I_1 &= \boldsymbol{\{U_1, (restart, null), \emptyset\}}\\
I_2 &= \{U_2, (click, w_1), (p_1, p_2)\}\\
I_3 &= \{U_3, (click, w_2), \emptyset\}\\
I_4 &= \{U_4, (back, null), \emptyset\}\\
I_5 &= \boldsymbol{\{U_5, (restart, null), \emptyset\}}\\
I_6 &= \{U_6, (long\ click, w_3), (p_3)\}\\
I_7 &= \{U_7, (click, w_4), \emptyset\}\\
I_8 &= \boldsymbol{\{U_8, (restart, null), \emptyset\}}\\
I_9 &= \{U_9, (click, w_2), \emptyset\}\\
I_{10} &= \{U_{10}, (back, null), \emptyset\}\\
I_{11} &= \boldsymbol{\{U_{11}, (END, null), \emptyset\}}
\end{aligned}$$

(a) Example of recorded exploration log which is used to identify relevant traces for evaluation

$$\begin{aligned}
T_1 &= \boldsymbol{\{U_1, (restart, null), \emptyset\}}\\
&\quad \{U_2, (click, w_1), (p_1, p_2)\}\\
&\quad \{U_3, (click, w_2), \emptyset\}\\
&\quad \{U_4, (back, null), \emptyset\}\\
&\quad \boldsymbol{\{U_5, (restart, null), \emptyset\}}\\
T_2 &= \boldsymbol{\{U_5, (restart, null), \emptyset\}}\\
&\quad \{U_6, (long\ click, w_3), (p_3)\}\\
&\quad \{U_7, (click, w_4), \emptyset\}\\
&\quad \boldsymbol{\{U_8, (restart, null), \emptyset\}}\\
T_3 &= \boldsymbol{\{U_8, (restart, null), \emptyset\}}\\
&\quad \{U_9, (click, w_2), \emptyset\}\\
&\quad \{U_{10}, (back, null), \emptyset\}\\
&\quad \boldsymbol{\{U_{11}, (END, null), \emptyset\}}
\end{aligned}$$

(b) Traces extracted from the exploration log which are used to measure *reproducibility* and *reachability* in the *Extract* and *Filter* steps

$$\begin{aligned}
T_1' &= \boldsymbol{\{U_1, (restart, null), \emptyset\}}\\
&\quad \{U_2, (click, w_1), (p_1)\}\\
&\quad \{U_3, (click, w_2), \emptyset\}\\
&\quad \{U_4, (back, null), \emptyset\}\\
&\quad \boldsymbol{\{U_5, (restart, null), \emptyset\}}\\
T_1'' &= \boldsymbol{\{U_1, (restart, null), \emptyset\}}\\
&\quad \{U_2, (click, w_1), (p_2)\}\\
&\quad \{U_3, (click, w_2), \emptyset\}\\
&\quad \{U_4, (back, null), \emptyset\}\\
&\quad \boldsymbol{\{U_5, (restart, null), \emptyset\}}\\
T_2 &= \boldsymbol{\{U_5, (restart, null), \emptyset\}}\\
&\quad \{U_6, (long\ click, w_3), (p_3)\}\\
&\quad \{U_7, (click, w_4), \emptyset\}\\
&\quad \boldsymbol{\{U_8, (restart, null), \emptyset\}}
\end{aligned}$$

(c) Traces with a single API to be replayed with *Replay with Enforcement* during *Restrict* step

Fig. 3. Recorded exploration log and traces extracted from it. Replay traces are extracted from the exploration and, once confirmed, they are split per widget and API to be restricted

was targeted by the action (if any). Our current types of interactions are: *click*, *long click*, *restart* (restart app) or *back* (press return button), however this model can be extended to support additional interactions. Finally, P is modeled as a set of monitored APIs ($p \in P$) which were invoked between the start of the action and its end (the moment the UI stabilizes again).

B. Extract

To accurately measure the impact of restricting an API invocation in an action, we first need to identify segments (traces) within the exploration which accessed sensitive resources. We denote a trace a sequence of interactions from L which start and end with an action of type *reset*[1]. A relevant trace is a trace where at least one interaction accessed a sensitive resource.

Each trace can be seen as a test case, where the AuT is started, a set of actions are executed and the AuT is closed. Assume the exploration log shown in Figure 3a. It starts with a reset action (I_1), clicks into 2 widgets (I_2, I_3), presses the back button (I_4), restarts the AuT (I_5), clicks on other 2 widgets (I_6, I_7), restarts again (I_8), presses one last widget (I_9), the back button (I_{10}) and terminates (I_{11}). In this example, there are three traces[2]: $T_1 = (I_1, I_2, I_3, I_4, I_5)$, $T_2 = (I_5, I_6, I_7, I_8)$ and $T_3 = (I_8, I_9, I_{10}, I_{11})$, which are illustrated in Figure 3b. The traces T_1 and T_2 access the sensitive resources p_1, p_2 and p_3 and are thus considered relevant. T_3 does not access any sensitive resource and is therefore removed.

C. Filter

Once the relevant traces are identified, it is necessary to evaluate if the traces can be replayed, that is, if each action in the trace can be re-executed – in a freshly installed AuT on an OS without previous app data – and it is possible to reach the same results as L.

This step is used to mitigate possible noise caused by changes in the app's behavior, such as advertisements being displayed at "random" places and times, or dynamic content. Moreover, it is also used to guarantee that there is not dependency between the traces.

A trace dependency happens when an action in a trace T_{i+1} can only be executed if an action in a trace T_i was previously executed. A concrete example of this problem can be observed in an Alarm app. Assume T_i created a new alarm and T_{i+1} would remove it. If T_i was not executed, the alarm to be removed in T_{i+1} would never be created, and thus T_{i+1} would not be reproducible. While it would be possible to solve trace dependencies by re-executing all traces ($T_0, \ldots, T_i$) before executing T_{i+1}, we opted to currently discard these traces to achieve improved performance speed ups.

In this step we each trace N times and evaluate each newly trace execution. We measure the similarity between the trace executions using 2 metrics: *reproducibility* and *reachability*.

Reproducibility, in this context, means the percentage of actions in the exploration trace which can be replayed. This metric represents how much of the trace could be successfully replayed. If the trace cannot be completely played it means that the app's behavior changes between executions, which would render any impact analysis faulty. A common example of traces which cannot be replayed are those where a monetization library randomly displays advertisement pop-ups on different moments. The following rules apply to replay an action:

> *Reproducibility measures how much of a recorded trace can be successfully re-executed.*

Rule 1 (Click and Long Click): A *click* or *long click* action can be replayed if the same widget actioned during recording is available and actionable in the current UI. Two widgets are considered the same if they are defined by the same data.

Rule 2 (Back): The press back action is used to return to a previously visited UI. To replay this action we ensure that the UI state before the executing the action is equal to

[1]We consider the last action in the exploration (*END* as a *restart* for trace generation purposes.

[2]The reset action between traces is duplicated because it finishes the previous trace and starts the new one.

the UI state in the original trace, i.e., all widgets found in the recorded UI are also found in the current UI, as well as no new widget is observed. We use the same metric as the *click* action to identify widgets.

Rule 3 (Restart): The reset action is used to close and reopen the application. It occurs only on a trace's boundaries, that is, when it starts and ends, thus, this action can always be replayed.

We use the *reachability* to evaluate if some information – albeit not triggered by the set of explored actions in this trace – is lost during replay. We measure this by evaluating if all unique widgets reachable (observed) during the initial exploration were also reachable in the trace. We define *reachability* as the percentage of widgets observed in the initial exploration which were also observed in the trace replay.

This metric is relevant because dynamic analysis is inherently incomplete and thus, it is possible that not all widgets have been explored in relevant traces. Therefore, it is not possible to accurately measure the impact of restricting access to a sensitive resource in app functionality in environments where different (non-equivalent) widgets are displayed in each execution.

Reachability measures how much non-explored functionality was lost.

In the end of this step, we define a trace as *confirmed* if it is completely reproducible (*reproducibility = 100%*) and if the same widgets can always be observed (*reachability = 100%*). Only *confirmed traces* are used in the next step.

D. Restrict

Once we have removed the non-reproducible traces we can measure the impact of restricting API access in app functionality by replaying each trace while enforcing privacy constraints. With this goal we propose *Replay with Enforcement*, a modified version of traditional replay approaches.

The goal of replay approaches is to reproduce a recorded test case (trace in our context). We used a replay approach in the *Filter* step, to evaluate if the traces could be accurately reproduced. In our *Replay with Enforcement* we *attempt* to replay as much of a trace as possible, while enforcing privacy constraints before specific actions.

To enforce constraints we instrument the AuT to add a proxy between the application and the OS, as illustrated in Figure 4. The dashed arrow in this image represents the AuT's expectation when invoking an OS provided API, i.e., that the API will be sent directly to the OS. Solid arrows denote the actual behavior of the instrumented AuT, where the proxy intercepts and forwards the API call to the OS.

To enforce a privacy constraint, the proxy component impersonates the OS by returning a default value for each data type, instead of accessing the OS resource and forwarding its return value. In Figure 4, this means the removal of the *Forwarded API call* arrow. We chose this approach to enforce

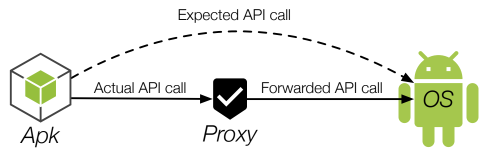


Fig. 4. Behavior of the proxy inserted into the application to enforce privacy policies

privacy constraints in order to not change the app's code nor the OS, allowing it to be used on default OS versions.

In order to measure the effect of restricting a specific API, each candidate trace should contain a single API otherwise it would not be possible to determine which effect was caused by which API, if a single trace blocks multiple APIs. Therefore, we replicate each trace X times – where X is the number of APIs contained in the trace – and each of these traces restricts a single API. Considering the traces T_1 and T_2 from Figure 3b as confirmed, T_1 would be replaced by T_1', restricting only p_1, and T_1'', restricting only p_2, as shown in Figure 3c.

The *Replay with Enforcement* then replays each action in the traces and, immediately before replaying the action in which the API invocation was recorded, it activates the proxy to restrict access to the sensitive resource. After the action was executed the proxy is again deactivated and the the access to the sensitive resource is allowed for the remaining functionality.

E. Impact Analysis

The last step of the approach is to produce a impact analysis by comparing the trace replayed in the *Restrict* step to the trace recorded in the original exploration.

During the *Filter* step we presented the metrics of *reproducibility* and *reachability*, which measured how much of the trace could be replayed and how much functionality was lost. For the impact analysis we reuse the *reachability* metric as previously defined and we adapt the *reproducibility* metric to account for behavioral changes caused by restricted access to sensitive resources.

The original *reproducibility* metric measured how much of the trace could be replayed. We redefine this metric to account only for the amount of actions which could be replayed *after* the restriction was enforcement. This change provides a more precise impact assessment because the *Filter* step guarantees that all actions before the restriction enforcement were the same.

Alongside a redefinition of the *reproducibility* metrics, we *extended* the rules which determined if an action can or cannot be replayed. The original rules were ensured to guarantee that the same exploration could be performed multiple times. These new rules aim to allow actions ot be replayed even when there are differences between the traces. The extensions are:

Rule 4 (Click and Long Click Pt. 2): Due to policy enforcement, behavioral changes may have presented widgets from being created or repositioned. Therefore, if the exact same widget cannot be found in the current UI, we look for a similar one. We define two widgets as similar if they possess the same values for all tuple attributes, except location. Only if a single widget satisfies this similarity definition the action can be replayed.

Rule 5 (Back Pt. 2): If the UI state before executing the action is different to the UI state which was recorded in the original trace, we evaluate if the next *Click* or *Long Click* action in the trace could be played in the current screen. If that action cannot be played, we consider this *back* action replay-able. This heuristic is based on the premise that the back button will mostly return to a previous UI and, if the next action can be played in the current UI it means that in a previous trace action a new UI was most likely not reached. Therefore, by pressing back in a state where the next action could be successfully executed, we risk invalidating all the remaining trace actions.

Using this expanded set of rules, we evaluate each trace explored during the *Restrict* step to measure the impact of restricting a specific API on a specific functionality in the app and we classify the impact as: *No Impact*, *Minor Impact* and *Major Impact*.

We define as *No impact*, traces which could be confirmed and replayed when enforcing policies without any visible effects on the exploration path (replay) or on visible functionality (available widgets). Traces are test cases, therefore, this category represents test cases which were successfully executed without warnings.

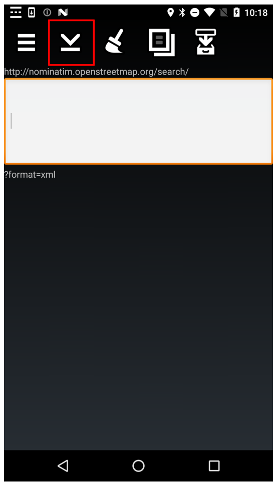
The *Minor impact* defines traces which could be successfully reproduced and replayed with policy enforcement, however, the API restriction led to the loss of previously available functionality. This category represents test cases which were successfully executed, i.e., performed all steps and reached the correct final state, but which ended with warnings. The exact impact cannot be measured in these traces because the missing functionality was not explored within it.

Major impact subsumes traces which could be confirmed but not completely replayed with restrictions, either due to missing or misplaced UI elements. It is analogous to having a test case which fails when restricting access to an API.

After the evaluation we produce an impact analysis regarding the restriction of one API on a specific program execution, containing how much of the trace could be replayed, which actions could not be executed and which widgets were missing.

Figure 5 illustrates an impact analysis report from a trace on the AAT (Activity Tracker) app[3] for a sensitive resource (*Socket.connect* API) in a specific widget (*Download* button of the UI to query a location provider). This impact analysis report states that, when restricting access to the API in the highlighted button only 87% of the subsequent actions could be replayed and 8% of the previously observed widgets can no longer be reached. The initial failure happened on a *back* action, with subsequent failures in the next 3 *clicks*. As well as the missing widget is a TextView with textual content "...".

[3]https://f-droid.org/packages/ch.bailu.aat/.



Trace A	
API:	**Socket.connect**
Widget:	**ViewGroup...**
Category:	**Major Impact**
# actions:	31
Confirm %:	100%
Replay %:	87%
Unseen %:	8%
Non replayable actions	
25 (Back)	-
26 (Click)	*ImageButton...*
27 (Click)	*ImageButton...*
28 (Click)	*ViewGroup...*
Unseen widgets	
Class	TextView
Resource ID	-
Text	...
Description	-

Location	x=526
	y=140
	width=263
	height=109

Fig. 5. Impact analysis extracted from with the proposed approach for the *Download* button of the *query OpenStreetMaps* UI for the Activity Tracker app

V. Implementation Details

[Nataniel: Should I add an implementation details to briefly document the tool, which as a contribution?]

VI. Empirical Evaluation

VII. Threats To Validity

VIII. Related Work

Different researches addressed privacy-related concerns on Android. It was possible to broadly categorize them into: OS modifications, app virtualization, static/dynamic analysis, test generation and blocking or mocking user data.

1) OS modifications to enhance privacy: within this category there are some approaches, such as TaintDroid [8], which monitor key interfaces (sources) and use dynamic tainting to identify sensitive data leaks on monitored sinks. Others, such as MockDroid [9], AppFence [10] and PDroid [11] modify the OS to allow users to control which information is shared with which apps. The major setback from these approaches is their reliance on OS modifications, which requires them to be adapted for each new operating system version. One last work in this category, which attempts to minimize this setback, is presented in [12], where minor changes to actual Android services are proposed in order to allow a fine-tuned permission control with minimal OS modification. Differently from these approaches, our analysis is performed only on application level and runs on a vanilla version of Android.

2) App Virtualization to separate the OS and the applications: allowing key interfaces to be monitored and malicious calls to be blocked and thus can be used for privacy enhancement. Cells [13], for example, presents a virtualization architecture that enable multiple virtual smartphones to run simultaneously on the same physical cellphone in an isolated manner. A similar approach is used by L4Android [14], which presents a generic OS framework that encapsulates the original

Android OS into a virtual machine, allowing highly secure applications to be run by alongside traditional ones. One last research that fits this category is BoxMate [15] – from where we extracted the exploration engine used in this work – which sandbox applications, preventing them from accessing any resource that was not identified during testing. Most of these approaches possess the same setbacks as the OS modifications, which we avoid by not performing any OS change.

3) Static/Dynamic analysis: aim to enhance privacy by identifying faults or malicious behavior in a controlled environment. Static analysis can be applied on different system components. Kirin [16] and CHABADA [5], for example, evaluated an app's manifest file to determine if the requested permissions are consistent with the user's expectations. Using the app's source code COPES [17] removed permissions which were granted but not required by the app. A more complex source code analysis is performed by FlowDroid [18], which statically track the information flow between sensitive sources and sinks to identify potentially potential privacy leaks. Dynamic analysis approaches, such as Andromaly [19], analyze real execution traces to identify potentially malicious behavior. In addition, hybrid techniques, combining static and dynamic analyses were also proposed. An example of this combination is AASandbox [20], a sandbox which performs both static and dynamic analysis on apps to automatically detect suspicious behaviors. A recent survey about pros and cons of static and dynamic analysis techniques is presented in [21]. Our approach falls into the dynamic analysis category and can also affected by under-approximation.

4) Test generation: This is a subcategory of dynamic analysis. While not primarily focused on privacy enhancement, these techniques are used to generate input values and trigger actual app behavior, which can then be used to identify privacy-violating behavior. These techniques currently apply one of the following strategies to generate input values: *fuzz testing*, *model-based testing*, and *systematic testing*. *Fuzz testing* tools, such as Monkey [22], create random chains of events and therefore can be used to test how robust and capable or error handling an app is. It is not suitable, however, to perform tasks that demand human intelligence due to its random nature. *Model-based testing* tools, such as CuriousDroid [23], infer an app model and use it to generate test cases. The techniques used to construct the models vary from GUI decomposition to finite state machines. *Systematic testing* tools, such as EvoDroid [24] and IntelliDroid [25], use different methods, such as symbolic execution, evolutionary strategies and search algorithms, to systematically explore an app. In approach we enhanced the *fuzz testing* tool DroidMate to enforce API restrictions.

5) Blocking or mocking user data: Researches in this category enhanced privacy by restricting an app's access to user data either by mocking or blocking its access. LBE Privacy Guard [26], limits the app access to sensitive resources by revoking permissions, which will make some apps unusable. A different approach is used by SRT AppGuard [27], which uninstall the original app and reinstall a modified version where resource access can be restricted. This approach inspired ArtHook, which was exploited in this work. Another way of limiting access to resources is to decompile apps, edit their source code and recompile them. An example of this approach is Aurasium [28], which automatically repackage Android applications to have sandboxing and policy enforcement abilities.

IX. Conclusion and Future Work

References

[1] M. Nauman, S. Khan, and X. Zhang, "Apex: extending android permission model and enforcement with user-defined runtime constraints," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. ACM, 2010, pp. 328–332.

[2] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 627–638.

[3] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in android apps with permission use analysis," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 611–622.

[4] SnoopWall Inc, "Flashlight apps threat assessment report," https://goo.gl/dif9PH, 2015, accessed: 2017-08-28.

[5] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 1025–1035.

[6] R. Herbster, S. DellaTorre, P. Druschel, and B. Bhattacharjee, "Privacy capsules: Preventing information leaks by mobile apps," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '16. New York, NY, USA: ACM, 2016, pp. 399–411. [Online]. Available: http://doi.acm.org/10.1145/2906388.2906409

[7] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "Madam: Effective and efficient behavior-based android malware detection and prevention," *IEEE Transactions on Dependable and Secure Computing*, 2016.

[8] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 5, 2014.

[9] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "Mockdroid: trading privacy for application functionality on smartphones," in *Proceedings of the 12th workshop on mobile computing systems and applications*. ACM, 2011, pp. 49–54.

[10] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "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*. ACM, 2011, pp. 639–652.

[11] P. H. Zhang, J. Z. Li, S. Shao, and P. Wang, "Pdroid: Detecting privacy leakage on android," in *Applied Mechanics and Materials*, vol. 556. Trans Tech Publ, 2014, pp. 2658–2662.

[12] S. Bugiel, S. Heuser, and A.-R. Sadeghi, "mytunes: Semantically linked and user-centric fine-grained privacy control on android," *Technical Report TUD-CS-2012-0226, Center for Advanced Security Research Darmstadt (CASED)*, 2012.

[13] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh, "Cells: a virtual mobile smartphone architecture," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 173–187.

[14] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter, "L4android: a generic operating system framework for secure smartphones," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 39–50.

[15] K. Jamrozik, P. von Styp-Rekowsky, and A. Zeller, "Mining sandboxes," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 37–48.

[16] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *IEEE security & privacy*, vol. 1, pp. 50–57, 2009.

[17] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, “Automatically securing permission-based software by reducing the attack surface: An application to android,” in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 274–277.

[18] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, “Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps,” *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[19] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, “Crowdroid: behavior-based malware detection system for android,” in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 15–26.

[20] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak, “An android application sandbox system for suspicious software detection,” in *Malicious and unwanted software (MALWARE), 2010 5th international conference on*. Nancy, Lorraine, France: IEEE, October 2010, pp. 55–62.

[21] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, “The evolution of android malware and android analysis techniques,” *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, p. 76, 2017.

[22] Google, “Ui/application exerciser monkey,” https://developer.android.com/studio/test/monkey.html, 2017, accessed: 2017-06-19.

[23] P. Carter, C. Mulliner, M. Lindorfer, W. Robertson, and E. Kirda, “Curiousdroid: automated user interface interaction for android application analysis sandboxes,” in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 231–249.

[24] R. Mahmood, N. Mirzaei, and S. Malek, “Evodroid: Segmented evolutionary testing of android apps,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 599–609.

[25] M. Y. Wong and D. Lie, “Intellidroid: A targeted input generator for the dynamic analysis of android malware.” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[26] B. Liu, J. Lin, and N. Sadeh, “Reconciling mobile app privacy and usability on smartphones: Could user privacy profiles help?” in *Proceedings of the 23rd International Conference on World Wide Web*, ser. WWW ’14. New York, NY, USA: ACM, 2014, pp. 201–212. [Online]. Available: http://doi.acm.org/10.1145/2566486.2568035

[27] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, “Appguard–enforcing user requirements on android apps,” in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2013, pp. 543–548.

[28] R. Xu, H. Saïdi, and R. J. Anderson, “Aurasium: practical policy enforcement for android applications.” in *USENIX Security Symposium*, vol. 2012, 2012.