Journal of Interconnection Networks
Vol. 17, No. 1 (2017) 1740002 (67 pages)
© World Scientific Publishing Company
DOI: 10.1142/S0219265917400023

World Scientific
www.worldscientific.com

Improving Smartphone Security and Reliability*

IULIAN NEAMTIU[†]
Department of Computer Science, New Jersey Institute of Technology, Newark, NJ 07102, USA
ineamtiu@njit.edu

XUETAO WEI[‡], MICHALIS FALOUTSOS[§], LORENZO GOMEZ[¶], TANZIRUL AZIM[§], YONGJIAN HU[§] and ZHIYONG SHAN[‖]
[‡] *University of Cincinnati, Cincinnati, OH 45220, USA*
[§] *University of California, Riverside, CA 92521, USA*
[¶] *University of California, Los Angeles, CA 90095, USA*
[‖] *University of Central Missouri, Warrensburg, MO 64093, USA*

Received 00 Month 2016
Revised 4 December 2016

Users are increasingly relying on smartphones, hence concerns such as mobile app security, privacy, and correctness have become increasingly pressing. Software analysis has been successful in tackling many such concerns, albeit on other platforms, such as desktop and server. To fill this gap, he have developed infrastructural tools that permit a wide range of software analyses for the Android smartphone platform. Developing these tools has required surmounting many challenges unique to the smartphone platform: dealing with input non-determinism in sensor-oriented apps, non-standard control flow, low-overhead yet high-fidelity record-and-replay. Our tools can analyze substantial, widely-popular apps running directly on smartphones, and do not require access to the app's source code. We will first present two tools (automated exploration, record-and-replay) that increase Android app reliability by allowing apps to be explored automatically, and bugs replayed or isolated. Next, we present several security applications of our infrastructure: a permission evolution study on the Android ecosystem; understanding and quantifying the risk posed by URL accesses in benign and malicious apps; app profiling to summarize app behavior; and Moving Target Defense for thwarting attacks.

Keywords: Mobile applications; android; security; program analysis; record-and-replay; monitoring; profiling; moving target defense.

*Material based on our prior publications in OOPSLA'13, OOPSLA'15, Globecom'15, MobiCom'12, ACSAC'12, Milcom'15.
[†]Corresponding author.

1. Introduction

Users are increasingly relying on smartphones for a variety of tasks.[37,38] For example, apps such as PayPal Here, Intuit GoPayment, Square Register allow mobile payments by users swiping a credit card on a smartphone or tablet that runs the app; FDA has approved smartphone ultrasound.[32] Aside from civilian use, smartphones are also used more and more in military domains, e.g., through the Nett Warrior[5] or Android Tactical Assault Kit (ATAK)[1] initiatives. However, with increased smartphone use, the potential for smartphones being unreliable, or attack targets, also increases. Moreover, users, developers and vendors have little insight into, and assurance of smartphone behavior. Therefore, concerns such as app security and reliability become increasingly pressing.[19,41,44,47,89]

Security concerns refer to confidentiality, integrity, and availability. Examples of confidentiality issues include the following. An user of a smartphone credit card reader might legitimately ask "is it safe to swipe my credit card on this stranger's smartphone?". Similarly, a merchant using that device might ask "Does the app send customers purchases or locations to advertisers, hence exposing me to liability?". Examples of integrity include making sure that someone's smartphone is not turned into a "bot" that takes commands from its peers or a Command-and-Control center hence subverting the intended use of the smartphone. Availability requires that phones and apps are usable, e.g., responsive, so that users can count on the phone and apps being ready at any moment.

Reliability means that smartphone software will operate as intended for a certain period of time.

Stakeholders. Besides end-users, software developers face difficulties as well, since tools for analysis and testing smartphone apps are only beginning to mature now. For example, a developer for the aforementioned smartphone payment app might have a difficult time proving to the credit card company that their app is secure. Military users's concerns are even more stringent: when smartphones are not secure, or unreliable in the tactical field, lives can be at stake. Finally, vendors are under pressure to guarantee that the pre-installed software (that comes with the phone) is secure and reliable, since this software runs my default and cannot be easily modified/uninstalled by users.

Our main idea and contribution consists of suite of approaches, implemented in tools, that allow all the aforementioned stakeholders to achieve, or benefit from, higher degrees of smartphone security and reliability.

To substantiate our claims, we present our prior results in the area of Android security and reliability. We begin with two infrastructure tools; these tools help improve Android reliability, and are building blocks for security analyses. First, in Section 2 we present a technique that allows apps to be explored automatically, which enables a wide range of dynamic analyses and automatic testing. Second, in Section 3 we present an approach for record-and-replay that allows app executions to be reproduced deterministically, which supports a wide range of reliability and security tasks (e.g., reproducing bugs or attacks).

Next (and forming the bulk of this paper) we discuss security. In Section 4 we perform a comprehensive study of the Android permission system to see if the ecosystem is moving "in the right direction" in regards to permissions and their use for security. We found that permissions have limited effectiveness, that "Dangerous" permissions are the largest set and growing, and that apps violate the principle of least privilege. In Section 5 we present a tool and study of risk, namely the risk apps are exposed to when they connect to websites; we found that "good" apps talk to "bad" websites, exposing the user to risk. In Section 6 we present an app profiling system that exposes app traits (behavior, traffic) which have many implications, from security to performance. Finally, in Section 7 we present a Moving Target Defense approach that offers increased resistance against known and unknown attacks, as demonstrated on a real attack and 12 widely-used Android apps.

2. Infrastructure and Reliability: Automated Exploration

Testing is key to reliability. Unfortunately, manual testing for Android leads to low coverage, hence insufficient testing, and consequently poor reliability. In this section we present our approach for generating app test cases automatically which achieve much better coverage than manual testing.

To facilitate test case construction and exploration for smartphone apps, several approaches have emerged. The Monkey tool[10] can send random event streams to an app, but this limits exploration effectiveness. Frameworks such as Monkeyrunner,[9] Robotium[39] and Troyd[56] support scripting and sending events, but scripting takes manual effort. Prior approaches for automated GUI exploration[6,7,76,82,99,101] have one or more limitations that stand in the way of understanding how popular apps run in their natural environment, i.e., on actual phones: running apps in an emulator, targeting small apps whose source code is available, incomplete model extraction, state space explosion.

For illustration, consider the task of automatically exploring popular apps, such as Amazon Mobile, Gas Buddy, YouTube, Shazam Encore, or CNN, whose source code is not available. Our approach A^{3}E can carry out this task, since we connect to apps running naturally on the phone. However, existing approaches have multiple difficulties due to the lack of source code or running the app on the emulator where the full range of required sensor inputs (camera, GPS, microphone) or output devices (e.g., flashlight) is either unavailable[8] or would have to be simulated.

2.1. *System Model*

We have chosen Android as the target platform as it is currently the leading mobile platform in the US[27] and worldwide.[51] We now describe the high-level structure of Android platform and apps; introduce two kinds of Activity Graphs that define the high-level workflow within an app; and define coverage based on these graphs.

2.1.1. *Android App Structure*

Android platform and apps. Android apps are typically written in Java (possibly with some additional native code). The Java code is compiled to a `.dex` file, containing compressed bytecode. The bytecode runs in the Dalvik virtual machine, which in turn runs on top of a smartphone-specific version of the Linux kernel. Android apps are distributed as `.apk` files, which bundle the `.dex` code with a "manifest" (app specification) file named `AndroidManifest.xml`.

Android app workflow. A rich application framework facilitates Android app construction, as it provides a set of libraries, a high-level interface for interaction with low-level devices, etc. More importantly, for our purposes, the application framework orchestrates the workflow of an app, which makes it easy to construct apps but hard to reason about control flow.

A typical Android app consists of separate screens named *Activities*. An activity defines a set of tasks that can be grouped together in terms of their behavior and corresponds to a window in a conventional desktop GUI. Developers implement activities by extending the `android.app.Activity` class. As Android apps are GUI-centric, the programming model is based on callbacks and differs from the traditional `main()`-based model. The Android framework will invoke the callbacks in response to GUI events and developers can control activity behavior throughout its life-cycle (create, paused, resumed, or destroy) by filling-in the appropriate callbacks.

An activity acts as a container for typical GUI elements such as toasts (pop-ups), text boxes, text view objects, spinners, list items, progress bars, check boxes. When interacting with an app, users navigate (i.e., transition between) different activities using the aforementioned GUI elements. Therefore in our approach activities, activity transitions and activity coverage are fundamental, because activities are the main interfaces presented to an end-user. For this reason we primarily focused on activity transition during a normal application run, because its role is very significant in GUI testing.

Activities can serve different purposes. For example in a typical news app, an activity home screen shows the list of current news; selecting a news headline will trigger the transition to another activity that displays the full news item. Activities are usually invoked from within the app, though some activities can be invoked from outside the app if the host app allows it.

Naturally, these activity transitions form a graph. In Fig. 1 we illustrate how activity transitions graphs emerge as a result of a user interaction in the popular Android app, Amazon Mobile. On top we have the textual description of users' actions, in the middle we have an actual screen shot, and on the bottom we have the activities and their transitions. Initially the app is in the Main Activity; when the user clicks the search box, the app transitions to the Search Activity (note the different screen). The user searches for items by typing in item names, and a textual list of items is presented. When the user presses "Go", the screen layout changes as the app transitions to the Search List Activity.

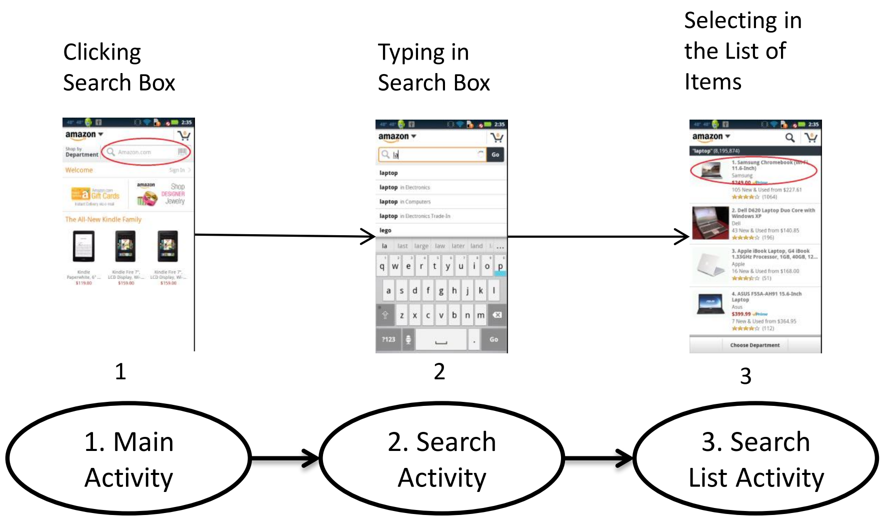

Fig. 1. An example activity transition scenario from the popular Android app, Amazon Mobile.

We now proceed to defining the activity transitions graphs that form the basis of our work.

2.1.2. *Static Activity Transition Graph*

The *Static Activity Transition Graph* (SATG) is a graph $G_S = (V_S, E_S)$ where the set of vertices, V_S, represents the app activities, while the set of edges, E_S, represents possible activity transitions. We extract SATG's automatically from apps using static analysis.

Figure 2 shows the SATG for the popular shopping app, Craigslist Mobile; the reader can ignore node and edge colors as well as line styles for now. Note that activities can be called independently, i.e., without the need for entering into another activity. Therefore, the SATG can be a disconnected graph. SATG's are useful for program understanding as they provide an at-a-glance view of the high-level app workflow.

2.1.3. *Dynamic Activity Transition Graph*

The *Dynamic Activity Transition Graph* (DATG) is a graph $G_D = (V_D, E_D)$ where the set of vertices, V_D, represents the app activities, while the set of edges, E_D, represents actual activity transitions, as observed at runtime.

A DATG captures the footprint of dynamic exploration or user interaction in an intuitive way and is a subgraph of the SATG. Figure 2 contains the DATG for the popular shopping app, Craigslist Mobile: the DATG is the subgraph consisting of solid edges and nodes. Paths in DATG's illustrate sequences of actions required to reach a particular state of an app, which is helpful for constructing test cases or reproducing bugs.

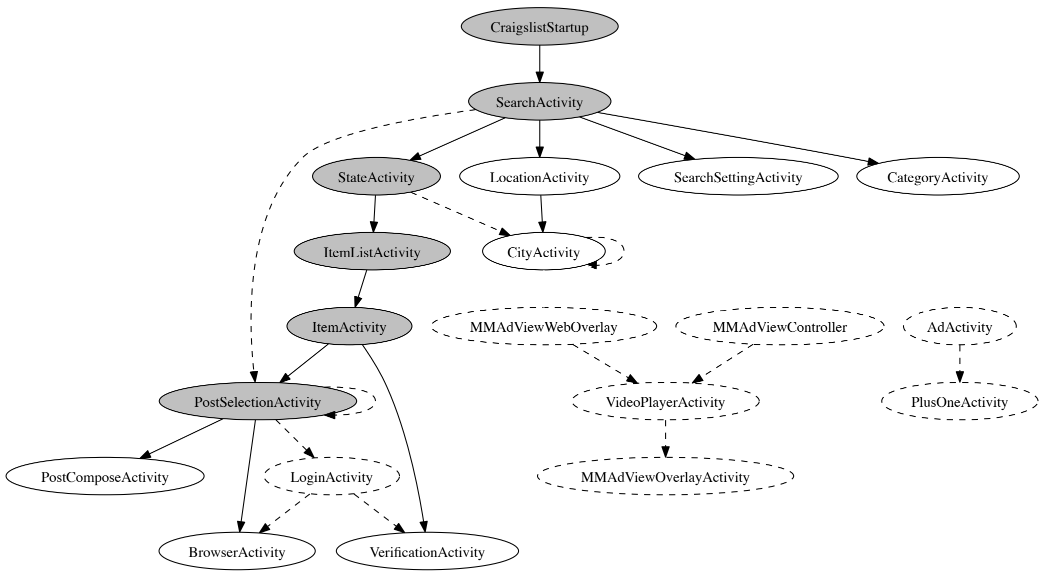

Fig. 2. Static Activity Transition Graph extracted automatically by our approach from the Craigslist Mobile app. Grey nodes and associated edges have been explored by users. Solid-contour nodes (grey or white) and solid-line edges were traversed dynamically by our exploration. Dashed-contour nodes and dashed-line edges remained unexplored. Activity names are simplified for legibility.

2.1.4. *Coverage Metrics*

We chose two coverage metrics as basis for measuring and assessing the effectiveness of our approach: activity coverage and method coverage. We chose these metrics because they strike a good balance between utility and collection overhead: first, activities and methods are central to app construction, so the numeric values of activity and method coverage are intuitive and informative; second, the runtime performance overhead associated with collecting these metrics is low enough so that user experience and app performance are not affected. We now proceed to defining the metrics.

Activity coverage. We define *activity coverage* (AC) as the ratio of activities reached during execution (AR) to the total number of activities defined in the app (AT), that is, $AC = \frac{AR}{AT}$. Intuitively, the higher the AC for a certain run, the more screens have been explored, and the more thorough and complete the app exploration has been. We retrieve the AR dynamically, and the AT statically.

Method coverage. Activity coverage is intuitive, as it indicates what percentage of the screens (that is, functionality at a high level) are reached. In addition, users might be interested in the thoroughness of exploration measured at a lower, method-level. Hence we use a finer-grained metric — what percentage of methods are reached — to quantify this aspect. We define *method coverage* (MC) as the ratio of methods called during execution (ME) to the total number of methods defined in the app (MT), that is, $MC = \frac{ME}{MT}$.

Table 1. Overview of our examined apps.

App	**Type**	**Category**	**Size**		**#Downloads**
			Kinst.	KBytes	
Amazon Mobile	Free	Shopping	146	4,501	58,745
Angry Birds	Free	Games	167	23,560	1,586,884
Angry Birds Space P.	Paid	Games	179	25,256	14,962
Advanced Task Killer	Free	Productivity	9	75	428,808
Advanced Task Killer P.	Paid	Productivity	3	99	4,638
BBC News	Free	News&Mag.	77	890	14,477
CNN	Free	News&Mag.	204	5,402	33,788
Craigslist Mobile	Free	Shopping	56	648	61,771
Dictionary.com	Free	Books&Ref.	105	2,253	285,373
Dictionary.com Ad-free	Paid	Books&Ref.	49	1,972	2,775
Dolphin Browser	Free	Communication	248	4,170	1,040,437
ESPN ScoreCenter	Free	Sports	78	1,620	195,761
Facebook	Free	Social	475	3,779	6,499,521
Tiny Flashlight + LED	Free	Tools	47	1,320	1,612,517
Movies by Flixster	Free	Entertainment	202	4,115	398,239
Gas Buddy	Free	Travel&Local	125	1,622	421,422
IMDb Movies & TV	Free	Entertainment	242	3,899	129,759
Instant Heart Rate	Free	Health&Fit.	63	5,068	100,075
Instant Heart R.-Pro	Paid	Health&Fit.	63	5,068	6,969
Pandora internet radio	Free	Music&Audio	214	4,485	968,714
PicSay – Photo Editor	Free	Photography	49	1,315	96,404
PicSay Pro – Photo E.	Paid	Photography	80	955	18,455
Shazam	Free	Music&Audio	308	4,503	432,875
Shazam Encore	Paid	Music&Audio	308	4,321	18,617
WeatherBug	Free	Weather	187	4,284	213,688
WeatherBug Elite	Paid	Weather	190	4,031	40,145
YouTube	Free	Media&Video	253	3,582	1,262,070
ZEDGE	Free	Personalization	144	1,855	515,369

We found that all the examined apps, except Advanced Task Killer, ship with third-party library code bundled in the app's APK file; we exclude third-party methods from ME and MT computations as these methods were not defined by app developers hence we consider that including them would be misleading. We measured the ME using runtime profiling information and the MT via static analysis.

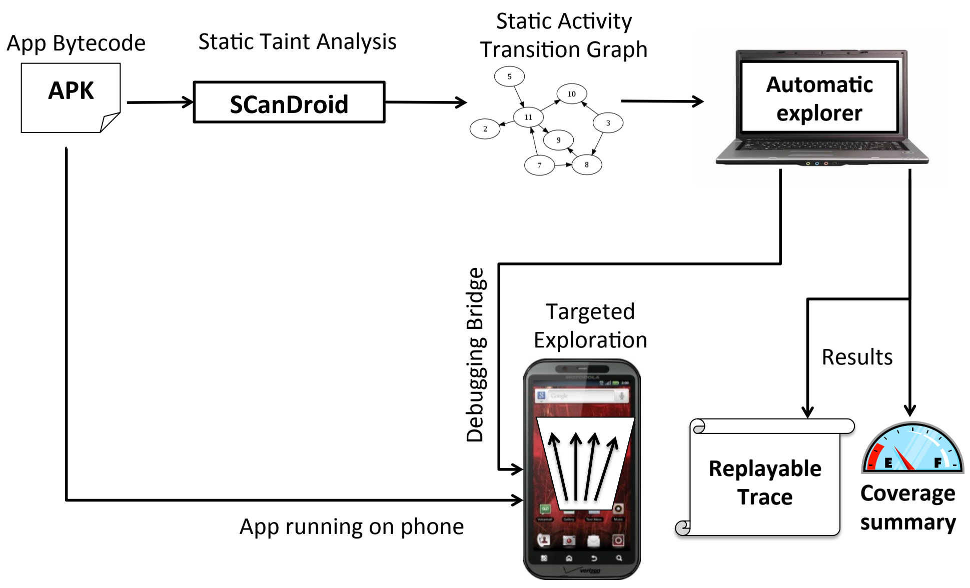

Fig. 3. Overview of Targeted Exploration in A^{3}E.

2.2. *Implementation*

To tackle these challenges, we have built the Automatic Android App Explorer (A^{3}E), an approach and open-source tool[a] for systematically exploring real-world, popular apps Android apps running on actual phones.[85] Developers can use our approach to complement their existing test suites with automatically-generated test cases aimed at systematic exploration. Since A^{3}E does not require access to source code, users other than the developers can execute substantial parts of the app automatically. A^{3}E supports sensors and does not require kernel- or framework-level instrumentation, so the typical overhead of instrumentation and device emulation can be avoided. Hence we believe that researchers and practitioners can use A^{3}E as a basis for dynamic analyses[28] (e.g., monitoring, profiling, information flow tracking), testing, debugging, etc.

To understand the level of exploration attained by Android app users in practice, we performed a user study and measured coverage during regular interaction. For the study, we enrolled 7 users that exercised 28 popular Android apps. We found that across all apps and participants, on average, just 30.08% of the app screens and 6.46% of the app methods were explored.[85]

Our approach for automated exploration is: given an app, we construct systematic exploration traces that can then be replayed, analyzed and used for a variety of purposes, e.g., to drive dynamic analysis or assemble test suites. Our approach consists of two techniques, *Targeted Exploration* and *Depth-First Exploration*. Targeted Exploration (Fig. 3) is a directed approach that first uses static bytecode

[a]http://spruce.cs.ucr.edu/A3E/

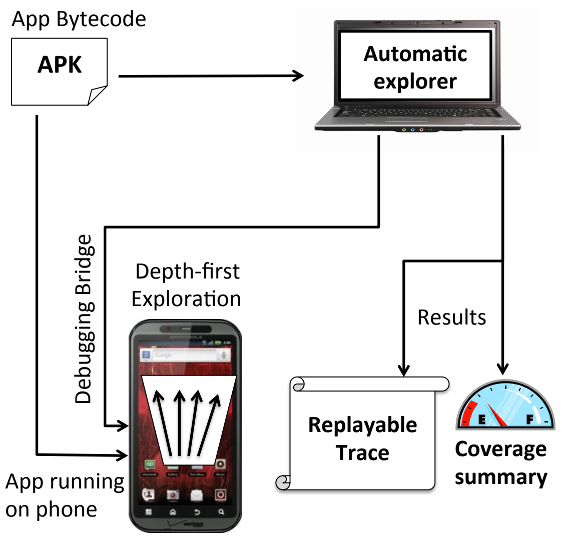

Fig. 4. Overview of Depth-first Exploration in A^{3}E.

analysis to extract a Static Activity Transition Graph and then explore the graph systematically while the app runs on a phone. Depth-First Exploration (Fig. 4) is a completely dynamic approach based on automated exploration of activities and GUI elements in a depth-first manner.

2.3. *Evaluation*

Manual exploration. To understand the level of exploration attained by Android app users in practice, we performed a user study and measured coverage during regular interaction. For the study, we enrolled 7 users that exercised 28 popular Android apps. We found that across all apps and participants, on average, just 30.08% of the app screens and 6.46% of the app methods were explored.[85]

Automatic exploration with A^{3}E. We have found that A^{3}E is effective: on 25 popular apps, on average it attains 64.11% and 59.39% activity coverage via Targeted and Depth-first Exploration, respectively (a 2× increase compared to what the 7 users have attained); it also attains 29.53% and 36.46% method coverage via Targeted and Depth-first Exploration, respectively (a 4.5× increase compared to the 7 users). Our approach is also efficient: average figures are 74 seconds for Static Activity Transition Graph construction, 87 minutes for Targeted Exploration and 104 minutes for Depth-first Exploration.[85]

3. Infrastructure and Reliability: Record-and-Replay

Record-and-replay helps both reliability and security. It increases reliability as replaying "buggy" execution allows bugs to be found and fixed. It increases security as it allows attacks to be replayed hence understand the loss of confidentiality or integrity.

To support record-and-replay, we have developed VALERA (VersAtile yet Lightweight rEcord and Replay for Android).[48] VALERA records and replays smartphone apps, by intercepting and recording input streams and events with minimal overhead and replaying them with exact timing.

While useful, record-and-replay on smartphones has proven difficult: smartphone apps revolve around concurrent streams of events that have to recorded and replayed with precise timing. To keep overhead low, prior record-and-replay approaches for smartphones only capture GUI input[58] which hurts accuracy as they cannot replay input from the network or sensors; or events, to reproduce event-based race.[65] Prior work on record-and-replay for desktop and server platforms[33,67] has relied on techniques such as hardware changes, or instruction logging which is too heavyweight for mobile apps. To address these challenges, we have developed VALERA (VersAtile yet Lightweight rEcord and Replay for Android),[48] a novel *sensor- and event-stream driven* approach to record-and-replay; VALERA is practical and has been designed to meet several key desiderata:

(1) *Support I/O (sensors, network) and record system information required to achieve high accuracy and replay popular, full-featured apps.*
(2) *Accept APKs as input — this is how apps are distributed on Google Play — rather than requiring access to the app source code.*
(3) *Work with apps running directly on the phone, rather than on the Android emulator which has limited support for only a subset of sensors.*
(4) *Low overhead to avoid perturbing the app's execution.*
(5) *Require no hardware, kernel, or VM changes.*

Our experiments shows that: VALERA is able to replay 50 popular Android apps which use a variety of sensors with low overhead (about 1% for either record or replay). VALERA is effective for reproducing bugs by replaying the input and event schedule that led to an error state. With the support of deterministic replay of event schdules, VALERA is able to reproduce hard-to-debug event-driven races.

3.1. *Overview*

VALERA consists of an API interception component and a runtime component. App instrumentation, achieved via bytecode rewriting, is used to intercept the communication between the app and the Android framework to produce log files. The runtime component is a manually instrumented Android Framework which is used to log and replay the event schedule.

Automatic interception through app rewriting. VALERA leverages Redexer[57] (an off-the-shelf Dalvik bytecode rewriting tool) to instrument the app. Given the original app (APK file) along with an *Interceptor specification*, VALERA is able to find all the callsites in the bytecode that match the specification and should be intercepted. The specification consists of a list of API methods along with simple annotations on how the methods and their parameters should be treated. Finally, the

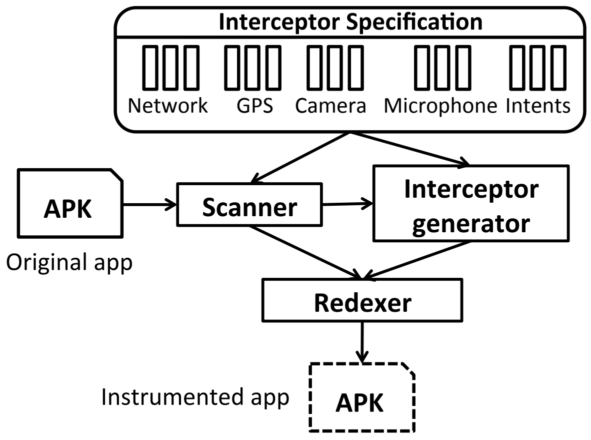

Fig. 5. Overview of VALERA's automatic interception.

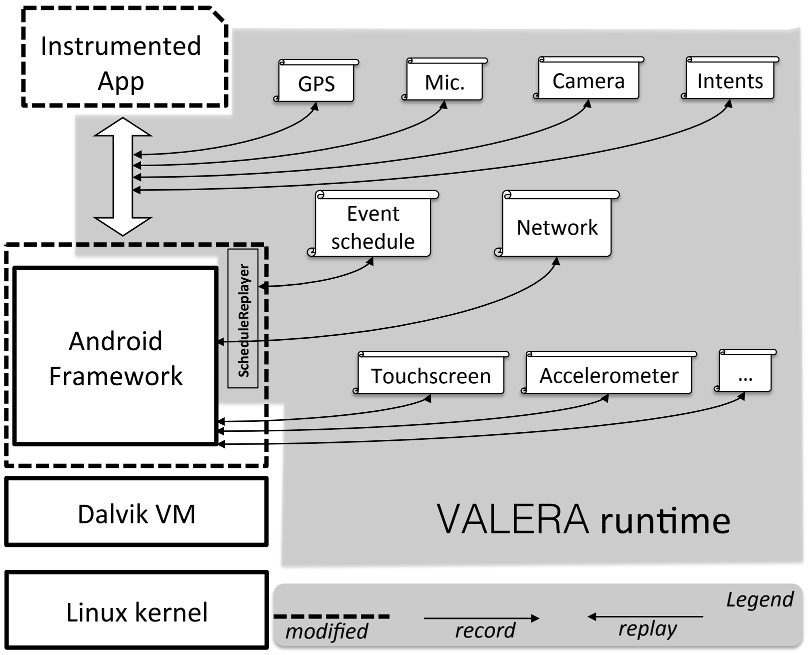

Fig. 6. Overview of the VALERA runtime.

dynamic intercepting modules and stubs are passed on to the *Redexer* to effect the interception, and repackages the bytecode into an instrumented APK. An overview is shown in Fig. 5.

The runtime component is shown in Fig. 6 (the grey area on the right). The instrumented app runs on top of the instrumented AF, which in turn runs on top of unmodified versions of the VM and the kernel. App instrumentation, achieved via bytecode rewriting as explained previously, is used to intercept the communication between the app and the AF to produce log files (values and timestamps) associated with network and high-level sensor input, such as GPS, microphone, and camera; intents are also intercepted at this point. AF instrumentation (which we performed manually) is used to log and replay the event schedule — see the ScheduleReplayer vertical box inside the AF. As the arrow directions indicate, during record the value/timestamp stream flows from left to right (toward the log files), and during replay from right to left (from the log files to the app/AF). To sum up, the VALERA runtime consists of record and replay code and the log files; this code runs inline in the app and AF, with no extra processes or threads needed. Other apps that execute concurrently run in their own address space, on their own VM copies; we omit them for clarity.

Recording and replaying the event schedule. Android is an event-driven system. Events can be classified into two groups: external and internal. External events come from the underlining hardware while internal events are posted by different threads. VALERA records the event schedule by logging each event and event processing operation into a trace file. Each event, either internal or external, is assigned a Lamport timestamp (logic order number) in the schedule. During replay, VALERA replays the events in the recorded order.

Effectiveness and efficiency. VALERA is effective: we were able to replay 50 popular apps (most of them have in excess of 10 million installs) that use a variety of sensors. Experiments show that VALERA imposes just 1.01% time overhead for record, 1.02% time overhead for replay, 208KB/s space overhead, on average, and can sustain event rates exceeding 1,000 events/second.

3.2. *Replay Applications*

VALERA has many applications in Android development; we present several.

Semantic sensor data alteration. To test the stability of an app, VALERA also provides features to alter the recorded sensor data in a semantically meaningful way. For example, VALERA can alter GPS readings (e.g., inject a null location object to simulate GPS hardware exceptions), blur or darken the pictures captured by camera to emulate different physical environments, or change the sample rate of the audio data to test the reaction of the app towards different audio qualities.[49]

Reproducing event-driven races. Event-driven races in Android are hard to debug and reproduce by current record-and-replay tools. With deterministic event order recorded, VALERA can help reproduce these races by preserving the exact event ordering. Our experiments show that VALERA can do this effectively on several open source apps: we were able to reproduce harmful event-driven races that crash the app (e.g., Tomdroid) or lead to incorrect GUI view state (e.g., NPR News).[50]

4. Security: Is the Ecosystem Moving in the "Right" Direction?

We now discuss how we can help improve Android security: specifically, we study Android's permission model, along with its security implications, advantages/disadvantages, and trends.

To ensure security and privacy, Android uses a permission-based security model to mediate access to sensitive data, e.g., location, phone call logs, contacts, emails, or photos, and potentially dangerous device functionalities, e.g., Internet, GPS, and camera. The platform requires each app to explicitly request permissions up-front for accessing personal information and phone features. App developers must define the permissions their app will use in the `AndroidManifest.xml` file bundled with the app, and then, users have the chance to see and explicitly grant these permissions as a precondition to installing the app. At runtime, the Android OS allows or denies use of specific resources based on the granted permissions. In practice, this security model could use several improvements, e.g., informing users of the security implications of running an app, revoking/granting app permissions without reinstalling the app, or moving towards finer-grained permissions.

In fact, the Android permission model attracts emerging malware that challenges the system to exploit vulnerabilities in order to perform privilege escalation attacks — permission re-delegation attacks,[13] confused deputy attacks, and colluding attacks.[78] As a result, users can have sensitive data leaked or subscription fees charged without their consent (e.g., by sending SMS messages to premium numbers via the SMS related Android permissions, as the well-known Android malwares Zsone and Geinimi do[97]). While most of these attacks are first initiated when a user downloads a third-party app to the device, to make matters worse, even stock Android devices with pre-installed apps are prone to exposing personal privacy information due to their higher privilege levels (e.g., the notorious HTCLogger app[11]).

We study the evolution of the Android ecosystem to understand whether the permission model is allowing the platform and its apps to become more secure.[92] Following a systematic approach, we use three different types of characterizations (third-party app permissions vs pre-installed app permissions, and two permission classifications from Google). We study multiple Android platform releases over three years, from *Cupcake* (April 2009) to *Ice Cream Sandwich* (December 2011). We use a stable dataset of 237 evolving third-party apps covering 1,703 versions (spanning a minimum of three years). Finally, we investigate pre-installed apps from 69 firmwares, including 346 pre-installed apps covering 1,714 versions. To the best of our knowledge, this is the first longitudinal study on Android permissions and the first study that sheds light on the co-evolution of the whole Android ecosystem: platform, third-party apps, and pre-installed apps.

Our overall conclusion is that the security and privacy of the ecosystem (platform and apps) do not improve, at least from the user's point of view. For example, the evolution moves more and more toward violating the *principle of least privilege*, a fundamental security tenet. Specifically, our study of the permission evolution of the Android ecosystem leads to the following observations:

The number of permissions defined in Android platform tends to increase, and the `Dangerous`-level set of permissions is the most frequent and continues to grow. There were 103 Android permissions in the first widely-used release (API level 3); the number of permissions has grown to 165 in the most current release (API level 15). Furthermore, the `Dangerous`-level permissions is always the largest group across all API levels, e.g., 60 out of 165 permissions in API level 15, and is still growing.

Added platform permissions cater to hardware manufacturers and their apps, rather than third-party developers. Nearly half (49.1% in API level 15) of all permissions are not accessible to third-party developers. Furthermore, of all the *added* permissions between API levels 3 to 15, most (49 out of 62) are in privilege levels that are not available to third-party developers, e.g., `Signature` and `signatureOrSystem` levels.

As shown in Table 2, `Dangerous` permissions are added in 5 out of 11 categories. Most of them are from personal data-related categories, e.g, `PERSONAL_INFO`, `STORAGE` and `ACCOUNTS`. We believe that this evolutionary trend shows that the Android platform provides more channels to harvest personal information from the device, which could increase the privacy breach risk if these permissions may be abused by Android apps.

Table 2. Added `Dangerous` permissions and their categories.

Dangerous permission	**Category**
READ_HISTORY_BOOKMARKS	Personal Info
WRITE_HISTORY_BOOKMARKS	Personal Info
READ_USER_DICTIONARY	Personal Info
READ_PROFILE	Personal Info
WRITE_PROFILE	Personal Info
READ_SOCIAL_STREAM	Personal Info
WRITE_SOCIAL_STREAM	Personal Info
WRITE_EXTERNAL_STORAGE	Storage
AUTHENTICATE_ACCOUNTS	Accounts
MANAGE_ACCOUNTS	Accounts
USE_CREDENTIALS	Accounts
NFC	Network
USE_SIP	Network
CHANGE_WIFI_MULTICAST_STATE	System Tools
CHANGE_WIMAX_STATE	System Tools

Android platform permissions are not becoming more fine-grained. Finer-grained permissions in Android, e.g., separating the advertisement code permissions from host app permissions,[70] have been advocated by security groups from both academia and industry.[29,64,87] The basis for finer-grained permissions is the *principle of least privilege*, i.e., giving apps the minimum number of permissions necessary to provide a certain level of service.

We investigated whether Android permissions are becoming more fine-grained over time. After carefully examining the Android permissions from API level 3 to 15, we observe that the permission changes do *not* tend towards becoming more fine-grained. We found only one possible example of a permission splitting in `READ_OWNER_DATA`. However, there is no indication that the two new permissions were specifically designed to replace the previous one, as shown in the first example of Fig. 7. Overwhelmingly, the permission changes indicate that the Android platform is giving more flexibility and control to the phone vendors. For example, as shown in Fig. 7, `SEND_SMS` and `PHONE_STATE` permissions exist in both API level 10 and 14, but the newly added Android permissions `SEND_SMS_NO_CONFIRMATION` and `READ_PRIVILEGED_PHONE_STATE` gives the app a higher privileged access to the device. Further, those higher privileged permissions are `signatureOrSystem` permissions, which can only used by vendor developers.

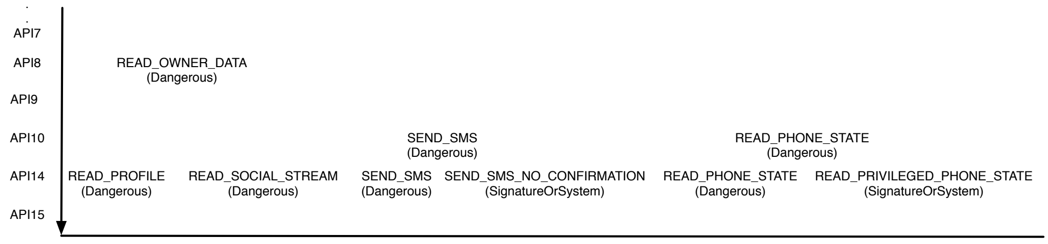
API7
API8
API9
API10
API14
API15
READ_OWNER_DATA (Dangerous)
SEND_SMS (Dangerous)
READ_PHONE_STATE (Dangerous)
READ_PROFILE (Dangerous)
READ_SOCIAL_STREAM (Dangerous)
SEND_SMS (Dangerous)
SEND_SMS_NO_CONFIRMATION (SignatureOrSystem)
READ_PHONE_STATE (Dangerous)
READ_PRIVILEGED_PHONE_STATE (SignatureOrSystem)

Fig. 7. Functionally-similar permissions added and deleted between API levels.

In other words, the platform does not seem to be moving towards more fine-grained permissions, which would in general be a step towards increased privacy or security. Instead, the permission changes indicate clearly that the Android platform is striving to give more flexibility and control to smartphone vendors, e.g., HTC, Motorola, Samsung, by providing them with permissions of higher privilege.

Permission additions dominate the evolution of third-party apps, of which `Dangerous` permissions tend to account for most of the changes. From the analysis of third-party apps, we found that the number of occurrences of adding Android permissions is significantly higher than the number of deleted permissions. Surprisingly, permission changes are not due to changes in the platform. Interestingly, among those additions, newer versions of apps tend to favor adding `Dangerous` permissions most often (66.11% of permission increases in apps consisted of at least one more `Dangerous` permission).

Macroscopic and microscopic patterns emerge when studying evolution of permission usage. We found evidence that `Dangerous` permission usage sometimes oscillate as an application evolves, which might indicate that developers are unclear about certain permission definitions, and their correct usage.

We analyzed the permissions added and deleted in the 1,703 versions of the 237 third-party apps in our stable dataset. We have found that most apps add permissions over time, with some apps adding more than 15 permissions. Only a small number of apps, about 10, delete permissions, and the deletions are limited to at most 3 permissions.

Table 3. App permission changes in the stable dataset.

	Total changes	**Induced by platform changes**
Add	857	14 (1.63%)
Delete	183	5 (2.73%)
Total	*1040*	*19 (1.82%)*

We present the total numbers of permission addition and deletion events in the stable dataset in Table 3: column 2 illustrates that the addition of permissions occurs much more frequently than the deletion of permissions. To disambiguate between genuine permission additions and additions induced by changes in the platform (e.g., as a result of added functionality), we also computed the permission changes induced by changes in the Android platform, which we show in column 3 of Table 3). Surprisingly, these induced changes only account for a small number of the permission changes: less than 3% of either additions or deletions. In sum, we were able to conclude that permission changes, which consist mostly of additions, are not due to changes in the platform.

We now set out to answer the question: *what is the primary cause for the permission additions*? We show the Top-5 most frequently added and dropped permission in the first column of Tables 4 and 5; column 2 of these tables will be explained shortly. For the added permissions, we found that Android apps became more aggressive in asking for resources, by asking for new permissions. For instance, the Android apps adopt permissions such as `WAKE_LOCK`, `GET_ACCOUNTS`, and `VIBRATE`. `WAKE_LOCK` prevents the processor from sleeping or the screen from dimming, hence allowing the app to run constantly without bothering the user for wake-up actions. `VIBRATE` enables the phone to vibrate for notifying the user when the corresponding apps invokes some functionality. In order to meet the increasing requirement of storage, `WRITE_EXTERNAL_STORAGE` is added to enable writing data into the external storage of the device such as an SD card. We note that permissions that do not improve the user experience, e.g., `ACCESS_MOCK_LOCATION` and `INSTALL_PACKAGES`, the apps simply drop them.

Table 4. Most frequently added permissions in the stable dataset.

Android permission	**In Top 20?**
ACCESS_NETWORK_STATE	✓
WRITE_EXTERNAL_STORAGE	✓
WAKE_LOCK	✓
GET_ACCOUNTS	×
VIBRATE	✓

Table 5. Most frequently deleted permissions in the stable dataset.

Android Permission	**In Top 20?**
ACCESS_MOCK_LOCATION	×
READ_OWNER_DATA	×
INSTALL_PACKAGES	×
RECEIVE_MMS	×
MASTER_CLEAR	×

Table 6. Top-20 most frequent permissions requested by malware.

Permission	**% of apps using it**
INTERNET	97.8%
READ_PHONE_STATE	93.6%
ACCESS_NETWORK_STATE	81.2%
WRITE_EXTERNAL_STORAGE	67.2%
ACCESS_WIFI_STATE	63.8%
READ_SMS	62.7%
RECEIVE_BOOT_COMPLETED	54.6%
WRITE_SMS	52.2%
SEND_SMS	43.9%
VIBRATE	38.3%
ACCESS_COARSE_LOCATION	38.1%
READ_CONTACTS	36.3%
ACCESS_FINE_LOCATION	34.3%
WAKE_LOCK	33.7%
CALL_PHONE	33.7%
CHANGE_WIFI_STATE	31.6%
WRITE_CONTACTS	29.7%
WRITE_APN_SETTINGS	27.7%
RESTART_PACKAGES	26.4%

As Android Apps are increasingly adding new permissions, users are naturally have security and privacy concerns, e.g., *how can they be sure that apps do not abuse permissions?*

For comparison, in Table 6 (from Zhou and Xiang[97]), we list the Top-20 permissions that Android malwares request (and abuse), as reported by Zhou and Xiang.[97] We now come back to column 2 in Tables 4 and 5; the columns show the result of comparing the added (and respectively, deleted) permissions in our stable dataset with the Top-20 malware permission list. A '✓' means the corresponding Android permission is in the Top-20 malware permission list, while a '×' means the permission is not in the list. We found that most of the added permissions are on the malware list, while none of the dropped permissions are on the list. Though we certainly can not claim these third-party apps are malicious, the trend should concern users: as apps gain more powerful access, the overall system becomes less secure. For example, in the *confused deputy* attack, a malicious app could compromise and leverage a benign app to achieve its malevolent goals .[78]

The characterization of permission changes we provided so far, in terms of absolute numbers (added/deleted), reveals the general trend toward apps requiring more and more permissions. In addition, we also performed an in-depth study where we looked for a finer-grained characterization of permissions evolution in terms of "patterns", e.g., repeated occurrences of permission changes.

Macro patterns. To construct the macro patterns, we use 0→1 and 1→0 as the basic modes, where '0' represents the state that the corresponding app does not use a particular permission, '1' represents the state that the corresponding app uses a particular permission, and '→' represents a state transition. In Table 8, we tabulate the macro-patterns we observed in the stable dataset, along with their frequencies. We found that the permission additions dominate the permission changes (0→1 has a 90.46% frequency), as pointed out earlier. We also found occurrences of other interesting patterns, e.g., permissions being deleted and then added back, though these instances are much less frequent.

Micro patterns. Some `Dangerous` permissions appear to be confusing developers. For example, the location permissions `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION`, provide different levels of location accuracy, on GSM/WiFi position and GPS location, respectively. Location tracking has been heavily debated because it could possibly be used to violate the user's privacy. We found that app developers handled the adding and deleting of these `Dangerous` location permission in an interesting way; to reveal the underlying evolution patterns of used by the `Dangerous` location permissions, we have done a case study of micro-patterns on two widely used location permissions, `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION`. We found that, although the most frequent macro evolution pattern of location permission is 0→1, the micro evolution patterns of the location permissions are quite diverse.

In Table 9, we tabulate the micro-patterns we observed for the location permission alone. For instance, 0→Both→Fine means both location permissions are used at first, then the ACCESS_COARSE_LOCATION permission is deleted in a later version of the app. 0→Fine→0→Fine shows the app added ACCESS_FINE_LOCATION at first, dropped it in a subsequent version, and finally, added back again. Though the table indicates several micro-patterns, note that using both location permissions dominates, with 50% of the total, which shows that more and more apps tend to include both location permissions for location tracking. We are able to make two observations. First, evolution patterns requesting Dangerous permissions clearly show the struggling balance between app usability and user privacy during the evolution of apps. Second, the patterns reveal that developers of third-party apps may be unclear with the correct usages of the Dangerous location permissions, which highlights the importance for the platform to be more clear on how to properly handle Dangerous permissions.

4.1. *Apps Want More Dangerous Permissions*

We now proceed to investigate the added permissions in the Dangerous protection level as they introduce more risks.

We found that 66.11% of permission increases in apps required at least one more Dangerous permission. In more detail, we list the frequently used Dangerous permissions in the first column of Table 7. We found that WRITE_EXTERNAL_STORAGE is the most requested Dangerous permission, in which sensitive personal or

Table 7. Frequently used Dangerous Android permissions of stable dataset.

Dangerous permission	**In Top 20?**
WRITE_EXTERNAL_STORAGE	✓
WAKE_LOCK	✓
READ_PHONE_STATE	✓
ACCESS_COARSE_LOCATION	✓
CAMERA	×
INTERNET	✓
ACCESS_FINE_LOCATION	✓
READ_LOGS	×
READ_CONTACTS	✓
RECORD_AUDIO	×
BLUETOOTH	×
CALL_PHONE	✓
CHANGE_WIFI_STATE	✓
GET_TASKS	×
MODIFY_AUDIO_SETTINGS	×
MANAGE_ACCOUNTS	×

enterprise files can be written to external media. This permission is also a hot-spot for most malicious activities. `INTERNET`, `READ_PHONE_STATE`, and `WAKE_LOCK` are also requested frequently by the new versions of the apps. The first two are needed to allow for embedded advertising libraries (ads), but these third-party ads are also raising privacy concerns of abusing the user's personal information. We then cross-checked this list with the Top-20 malware permissions,[97] as shown in column 2 of Table 7. We observed that 9 of the 16 frequent permissions listed are also frequently used by malicious apps. This significant overlap intensifies our privacy and security concerns.

An increasing number of apps are violating the principle of least privilege. The tendency of developers to request permissions that their apps do not need causes an app to become overprivileged (as is the case for 44.8% of apps).

Extra permission usage may lead to overprivilege, a situation in which an app requests the permission, but never uses the resource granted. This could increase vulnerabilities in the app and raise concern of security risks. In this section, we investigate the privilege patterns to determine whether Android apps became overprivileged during their evolution.

To detect overprivilege, we ran the Stowaway[14] tool on the stable dataset (1,703 app versions). We found that 19.6% of the newer versions of apps became overprivileged as they added permissions, and 25.2% of apps were initially overprivileged and stayed that way during their evolution. Although the overall tendency is towards overprivilege, we could not ignore the fact that 11.6% of apps decreased from overprivileged to legitimate privilege, a positive effort to balance usability and privacy concerns.

In addition, similar to the evolution patterns of permission usage, we also study the evolution patterns of overprivilege status for each app.[92] We found that the patterns Legitimate→Over and Over→Legitimate dominate at 58.57% and 32.14%, respectively. However, like in the patterns of permission usage, we also found other diverse patterns during the evolution of apps, which again shows that there may be confusion for third-party developers when deciding on what permissions to use for their app.

Finally, we have observed that `Dangerous` permissions are the major source that causes an app to be overprivileged, which again emphasizes that developers should exercise more care when requesting `Dangerous` permissions.

The power and privilege of pre-installed apps is growing. Sixty-six percent of pre-installed apps are overprivileged. Furthermore, pre-installed apps have more power to control and customize Android devices through Android platform-defined and self-defined higher protection level permissions, e.g., `Signature`- and `SignatureOrSystem`-level permissions. Though granting vendors higher privilege is not surprising, end-users (the actual owners of the device) still have security concerns.[11,63] We argue that since pre-installed apps have greater power over the

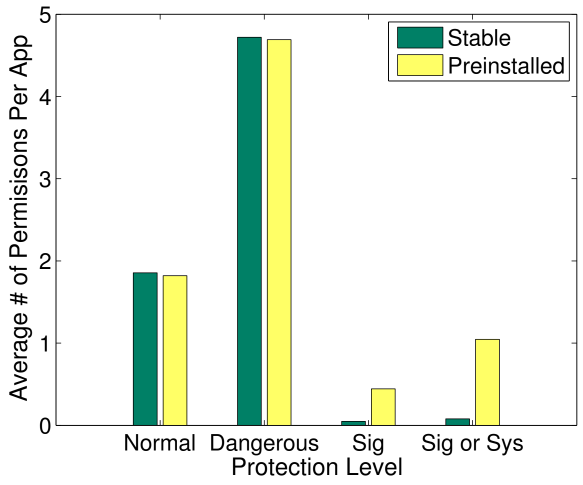

Fig. 8. Average number of permissions per app, for each protection level, from stable and pre-installed datasets.

device, the developers of pre-installed apps must understand and accept their responsibility to protect the end user.

Pre-installed apps have access to a richer set of higher-privileged permissions, e.g., at the `Signature` and `signatureOrSystem` levels, compared to third-party apps, which gives pre-installed apps access to more personal information on the device.[63] Thus, we should investigate how Android permissions are used in pre-installed apps. We conducted a permission-change analysis for pre-installed apps in a manner similar to the stable dataset. We found that 62.61% of pre-installed apps do not change their permissions at all, which is significant when compared to our third-party apps with only 15.68%. Further, pre-installed apps request many more `Signature` and `signatureOrSystem` level permissions than third-party apps, while at the same time requesting nearly just as many `Normal` and `Dangerous` level permissions. This shows that pre-installed apps have a much higher capability to penetrate the smartphone. Interestingly, the vendors also have the ability to define their own permissions inside the platform when they customize the Android platform for their devices. For example, HTC defines its own app update permission, `HTC_APP_UPDATE`.

The power of pre-installed apps requires great responsibility by vendors to ensure that this power is not abused. On one hand, vendors are able to customize pre-installed apps to take full advantage of all the hardware capabilities of the device, as well as create a brand-personalized look-and-feel to enhance user experience. On the other hand, users cannot opt out of pre-installed apps, and in most cases, cannot uninstall the pre-installed apps, which raises the question: *why should users be forced to trust pre-installed apps?* Hindering that trust is our finding that, despite being developed by vendors, 66.1% of pre-installed apps were overprivileged.

What if the power of pre-installed apps is used against the user with malicious intent? For example, the marred pre-installed app `HTCLogger` and other reported security compromised apps have already indicated such security risks do exist and can significantly damage the smartphone and/or the user data.[11,63] The vendors' `Signature` and `signatureOrSystem` level permissions can be exploited by malicious apps to do an array of damaging actions, such as wiping out user data, sending out SMS messages to premium numbers, recording user conversations, or obtaining the device location data of the device.[63]

As we analyzed the evolution of Android platform permissions, it was interesting to see the evolution trends benefit vendors, rather than users. With the power vendors have in pre-installed apps, developers of pre-installed apps should be more careful in their development as they represent the trusted computing base (TCB) of the Android ecosystem. Up until now, there has not been any clear regulations or boundary definitions that protect the user from pre-installed apps. We argue that, since pre-installed apps have more power and privilege over Android devices, vendors need to realize their responsibility to protect the end-user.

5. Security: URL Risk

Apps are very Internet-centric: a lot of their operation consists of accessing, downloading data from, or uploading data to, the Web via URLs[b]. Unfortunately, many websites are not trustworthy, e.g., they can be used to host malware, hence URLs have a certain risk factor associated with them. In this section we define and characterize this URL risk that affects Android apps.

Apps collect substantial amounts of information about users by mans of sensors that offer a wide range of context-sensitive functionality, from GPS- and compass-assisted navigation to song recognition to exercise tracking to picture geo-tagging and sharing. This is illustrated in Fig. 9: the private data (e.g., location, phone state, list of contacts) can be leaked by apps to all kinds of websites:[c] good, bad, or somewhere in-between.

We focus on a specific security and privacy question: *Do good apps talk to "bad" websites?* We use the term *bad websites* to refer to hosts and domains that have been labeled as inappropriate by malware repositories such as VirusTotal[3] and trustworthy reputation engines such as Web-Of-Trust (WOT).[4] In general, these bad websites engage in dangerous or annoying activities that range from distributing malware, to phishing to overly aggressive ads and spamming, as we discuss in Section 5.1. Adopting the terminology from WOT, we define the terms: (a) **malicious** website, implicated in distribution of malware, (b) **bad** website, that appears in blacklists, and (c) **low-reputation** websites that have a user rating of less than 60, with each

[b]https://en.wikipedia.org/wiki/Uniform_Resource_Locator
[c]Websites, domains, hosts and entities are used interchangeably in our paper.

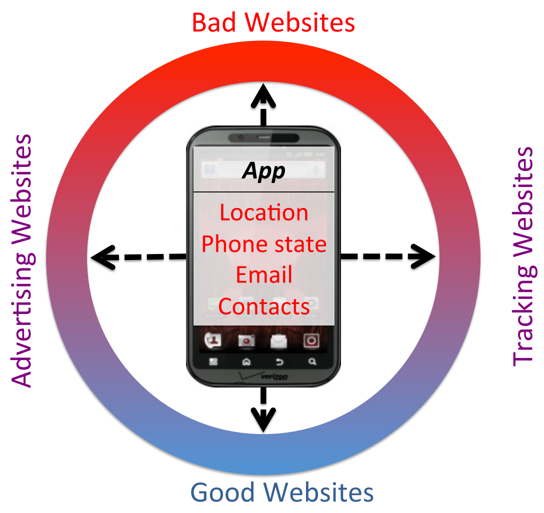

Fig. 9. Even good apps communicate with websites of variable reputation, which can raise a range of concerns.

category including the previous in the order presented. For labeling a website, we rely on information and the lists from WOT and VirusTotal, which are widely-used and widely considered as reference sources.

We focus on apps' network communication, since it is an obvious vector for security attacks: Internet access is a *de facto* capability for almost all apps. On the Android platform, most apps request Internet permission, while all apps in the iOS App Store have Internet access by default without even asking the user. It is also highly unlikely that apps will refrain from getting Internet access: (a) many apps needs access to the Internet to operate, and (b) many apps, especially the free ones, seek revenue by either showing ads or collecting information about the user and her behavior, such as mobility patterns. So the question is: does Internet access pose concerns for security and privacy, even for good apps? We use the term *good apps* here loosely to refer to apps that come from reputable developers, and widely vetted by large numbers of users. Interestingly, even identifying which websites an app talks hides several subtleties.

We have developed AURA (Android Url Risk Assessor), a systematic approach to identifying security and privacy concerns for apps based on the websites that an app talks to.[93] The first step is to comprehensively identify all such websites for a given app, which is non-trivial. We propose and compare the use of both dynamic and static analysis, and we argue that static analysis is necessary as many embedded websites are not contacted, even during exhaustive test runs. Second, we provide a taxonomy of websites using both malware detection and crowdsourcing efforts to capture a wide range of annoying or dangerous activities. We study 13,500 popular free Android apps from Google Play[2] that connect to 254,022 URLs and 1,260 malicious Android apps[97] that connect to 19,510 URLs.

Table 8. Macro evolution patterns of permission usage in the stable dataset.

Macro pattern	**Frequency**
$0\rightarrow 1$	90.46%
$1\rightarrow 0$	8.59%
$1\rightarrow 0\rightarrow 1$	0.84%
$1\rightarrow 0\rightarrow 1\rightarrow 0$	0.11%

AURA *overview.* Figure 10 presents the high-level architecture of AURA. Given an Android app, we use static analysis to extract the URLs embedded in the app. We also have a dynamic analysis component, which can be used selectively, as we did for a set of sample apps, to complement the static analysis. can scale well but it might miss URLs (e.g., due to redirection), hence has the need for dynamic analysis. With the URLs at hand, AURA performs classification and risk evaluation on both URLs and domains, with the help of VirusTotal and WOT. Finally, the output is a set of potential risks, in terms of bad websites the app talks to: malware, phishing, low-reputation websites. These risks are presented to the user.

The results of our work can be summarized in the following points.

a. Developing AURA. We develop a systematic and comprehensive approach focusing on a lesser-studied security aspect of apps. A key novelty is the use of both static and dynamic information: we use both static (bytecode) analysis and dynamic (execution) analysis when available. We employ widely-used classification labels for the bad websites, in order to make reporting of results consistent with industry standards.

b. The importance of static analysis. We show that dynamic analysis cannot match the thoroughness of static analysis. Even when apps are explored thoroughly (on average for two hours each) using high-coverage automated tools, dynamic analysis identifies less than half the URLs static analysis does. This suggests that using static analysis provides significant insight into an app's potential communication.

Dynamic analysis critically hinges on the availability of a good (high-coverage) testing suite, so that all the facets of an app can be explored. As argued above, even sophisticated tools such as A^{3}E have limited reach in terms of how thoroughly an app is explored. In the following, we demonstrate the effectiveness of static analysis when compared with dynamic analysis. In order to ensure representative results, we selected our test apps by following rigorous criteria as we did in previous work.[94] In Table 10, we present a comparison of the two analyses for each app, and over all apps. The first column contains the app name. The second column shows the exploration time required by dynamic analysis. Note that dynamic exploration is thorough, with apps being explored on average for *two hours*, which is far longer than the typical average app user session (71.56 seconds).[22] The third column shows the total number of URLs discovered by dynamic analysis for each app; on average,

Table 9. Micro evolution patterns for the location permissions; Fine represents the `ACCESS_FINE_LOCATION` permission, Coarse represents the `ACCESS_COARSE_LOCATION` permission, and Both means both Fine and Coarse are used.

Micro pattern	**Frequency**
Both	6.67%
Fine→Both	10.00%
Fine→Coarse	3.33%
Coarse→Both	10.00%
0→Both	20.00%
0→Fine	10.00%
0→Coarse	26.70%
0→Fine→Both	3.33%
0→Both→Fine	3.33%
0→Both→Coarse	3.33%
0→Fine→0→Fine	3.31%

6.1 URLs per app. The fourth column shows the number of URLs discovered via dynamic analysis that could not be found via static analysis. The last two columns show the number of URLs discovered by static analysis (total and the extra URLs compared to dynamic analysis). We observe that static analysis finds on average 11.9 domains that dynamic analysis does not find, whereas dynamic analysis finds on average 2.8 domains that static analysis does not. Therefore, we chose static analysis as our preferred method for performing the rest of the study. An additional advantage of static analysis is scalability: as URL extraction and classification takes on the order of seconds per app, static analysis is particularly suitable for analyzing large sets of apps.

c. Good apps talk to bad websites. We find that good apps can be interacting with questionable websites: for our examined apps, 8.8% communicate with malicious websites, 15% talk to bad websites, 73% with low-reputation websites (as defined above), and 74% of the apps talk to websites containing material not suitable for children.

d. Understanding bad intentions. We find the following intentions of bad websites: 43% of bad websites try to phish sensitive personal information or confidential financial information, e.g., credit card details, while 42% of bad websites are used for distribution of rootkits, trojans, viruses, malware, spyware, rogues and adware, and creating virus attacks. The rest of the bad websites, which account for 15%, intrusively and aggressively sell ads. These intentions vary from harming devices to stealing confidential data to annoying users.

Table 10. Summary of static and dynamic analyses; "extra" refers to domains found by one analysis but not the other.

	Dynamic analysis			**Static analysis**	
App	Time (minutes)	Total URLs	Extra URLs	Total URLs	Extra URLs
Amazon	131	5	2	10	6
AdvncdTaskKiller	47	0	0	3	3
AdvncdTaskKiller($)	58	0	0	0	0
BBCnews	52	13	8	5	2
CNN	161	8	6	12	10
Craigslist	91	7	3	7	3
Dictionary.com	131	19	15	13	8
Dictionary($)	156	0	0	13	13
Dolphin Browser	179	6	0	14	12
ESPN	44	1	1	7	7
Flixster	219	9	4	20	16
IMDB	126	4	3	17	17
InstantHeartRate	51	0	0	27	27
InstantHeartRate($)	49	0	0	27	27
Pandora	111	3	0	19	19
Picsay	121	0	0	3	3
Picsay($)	129	0	0	3	3
Shazam	239	12	4	24	20
Shazam($)	230	12	4	24	20
Weatherbug	107	14	3	16	16
Weatherbug($)	124	14	1	16	16
ZEDGE	114	8	8	16	15
Average	*121*	*6.1*	*2.8*	*13.4*	*11.9*

5.1. *Evaluation Results*

Our study is based on a sizable number of apps, both benign and malicious. After we apply AURA to these apps, we obtain 254,022 URLs from 13,500 benign apps and 19,510 URLs from 1260 malicious apps. In the remainder of the paper, with the exception of Section 5.1.2 where we discuss malicious apps, all the findings are based on analyzing the 254,022 URLs that the 13,500 benign apps talk to.[79] Note that the total number of URLs was obtained by adding the number of URLs for each app, so the dataset contains duplicates; we will discuss this next.

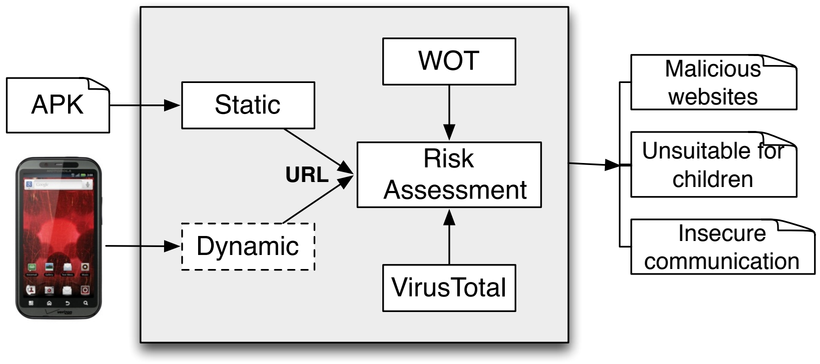

Fig. 10. AURA architecture.

5.1.1. *Malicious URLs*

Malicious sites could be visited or interacted with when using Android apps, which poses great risk especially when more and more apps harvest the personal information stored on the smartphone. We examined whether malicious URLs are used in our benign app dataset. To find such URLs, we cross-checked against VirusTotal's database. We found that 286 URLs were malicious; these URLs spread across 1,187 apps (8.8%). We believe that the 8.8% percentage is a significant source of concern. In Section 5.1.3 we present a detailed analysis of these blacklisted domains.

Table 11. Top 20 domains used in apps.

1	admob.com	11	tapjoyads.com
2	android2020.com	12	mydas.mobi
3	twitter.com	13	adwhirl.com
4	facebook.com	14	w3.org
5	airpush.com	15	wikipedia.org
6	google.com	16	amazonaws.com
7	android.com	17	psesc.com
8	gstatic.com	18	inmobi.com
9	mobclix.com	19	paypal.com
10	flurry.com	20	hubblesite.org

We now turn to investigating the domains (or hosts) in our dataset, that is the trustworthiness of sites without regard to the specific path. We found that 66% of the apps talk to at least one domain that has very poor reputation; that 74% of the apps talk to websites containing material not suitable for children; that malicious apps do not necessarily talk to ill-reputed websites; and that 15% of apps talk to blacklisted domains. We also found that Android apps tend to have more tracking services than advertisement services.

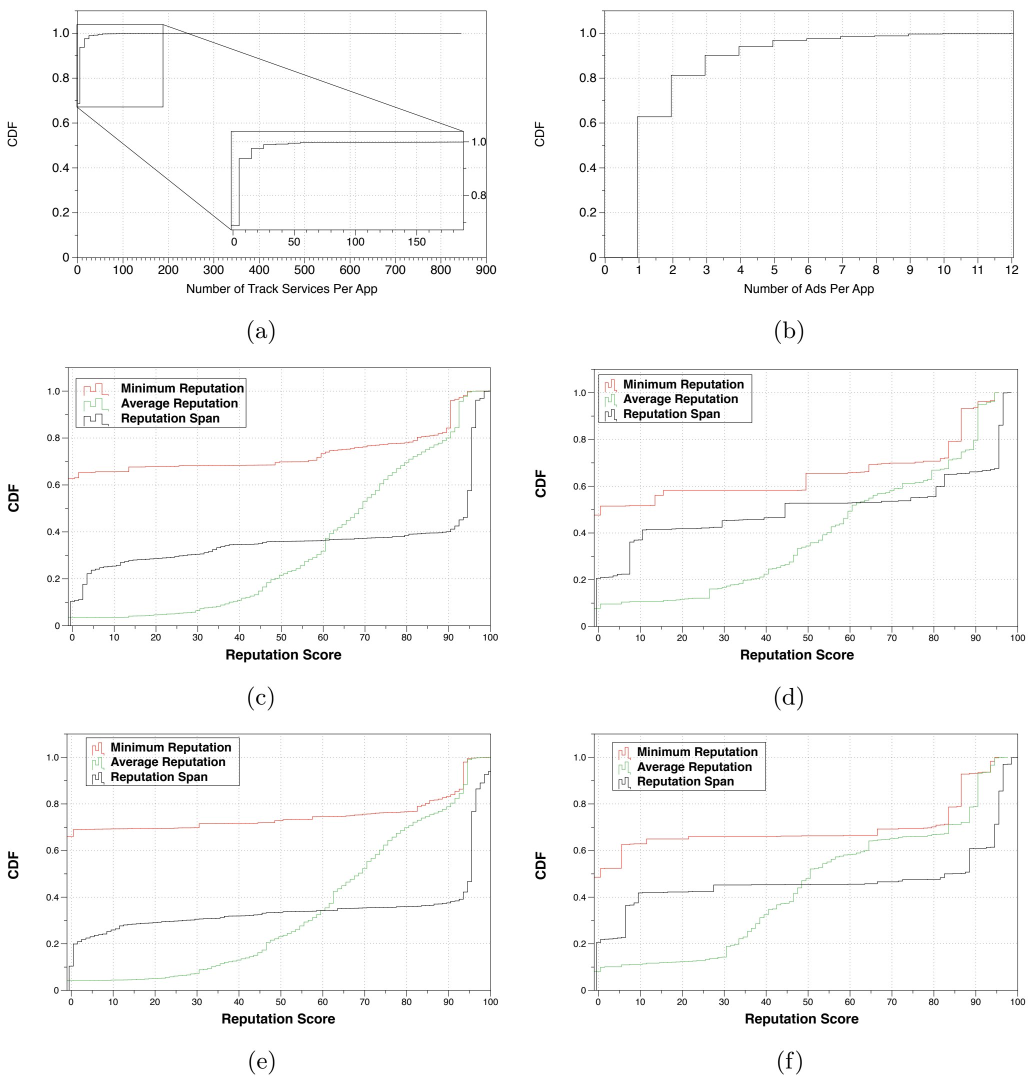

Fig. 11. (Color online) (a) Distribution of number of tracking services per each app; (b) Distribution of number of advertisement services per each app; (c) Domain reputation distribution in benign apps; (d) Domain reputation distribution in malicious apps; (e) Child safety reputation distribution in benign apps; (f) Child safety reputation distribution in malicious apps.

We also computed the top domains used in the 13,500 benign Android apps, and present the top-20 in Table 11. We can see that advertisements (e.g., admob.com, flurry.com, airpush.com, inmobi.com), cloud services (e.g., amazonaws.com), social networking services (e.g., twitter.com, facebook.com) and payment solution services (e.g., paypal.com) are used intensively among apps. We also plot the distributions about the number of tracking and advertisement services of each app in Figs. 11(a) and 11(b). We can see that both services penetrate Android apps broadly, while Android apps tend to have more tracking services than advertisement sources.

5.1.2. *Trustworthiness and Child Safety*

We now investigate the reputation of the domains extracted from our URL dataset with respect to trustworthiness and child safety. WOT assigns each domain a reputation score from 0 (very poor reputation) to 100 (very good reputation). We use WOT's domain reputation score to compute the reputation for each domain in our dataset. If we could not find the reputation for a domain in WOT's database, we just assign that domain a reputation score of -1. For each app, we compute several reputation indicators: a *minimum reputation*, that is the lowest reputation score across all the domains used in the app; an *average reputation*, that is the average reputation score across all the domains used in the app; and a *reputation span*, that is the difference between the minimum and maximum reputation score across all the domains used in the app.

Benign apps. Our analysis has revealed several reasons for concern. We found that 63% of apps talk to at least one domain for which WOT does not have any reputation score. In addition, 73% of the apps talk to at least one domain that has unsatisfactory reputation (score is less than 60). In detail, 68% of the apps talk to domains with poor reputation (score is less than 40) and 66% of the apps talks to the domains with *very poor reputation* (score less than 20). Unsurprisingly, these trends are also reflected in the reputation span: 60% of the apps have reputation spans exceeding 90 points, meaning the apps mix high-reputation with low-reputation domains. These findings indicate that there is significant cause for concern even for benign apps: when these supposedly benign apps send information to low-reputation domains, users can be exposed to privacy and security risks.

Children use Android apps for many activities, e.g., gaming or social networking. Hence we proceed to analyze how child-safe the domains are, based on WOT's definition of child-safe domains. Similar to trustworthiness, we plot the child-safety reputation score for benign apps in Fig. 11(e). We observe that 74% of the benign apps talk to at least one domain that has unsatisfactory reputation based on user ratings, hence may not be suitable for children (for example, the website contains adult material). We believe that such findings could help Google Play, the main Android app marketplace, to better regulate app distribution in order to safeguard child safety.

Malicious apps. Intuitively, we would assume that malicious apps would contain low-reputation of domains. We do the same reputation evaluation for the malicious apps, and we find that our intuition would be wrong — malicious apps have similar distribution of trustworthiness and child safety as benign apps. For example, in terms of trustworthiness Fig. 11(d) indicates that about 55% of the apps have reputation less than 15, and there are fewer apps with large reputation spans. Child safety reputations (Fig. 11(f)) are also similar to benign apps.

This is unsurprising, since most malicious apps are created by injecting a malware veneer in a benign app via repackaging.[104] The edge that AURA provides is the ability to examine the reputation of domains the app talks to: it is important to tackle Android app security not only via traditional security techniques (that protect devices against technical threats such as viruses and other harmful software), but also via crowdsourcing. Hence our AURA approach can help protect the device and the app against threats that only the human eye can identify, such as scams, unreliable web stores and questionable content.

5.1.3. *Blacklisted Domains*

Blacklisted domains are known for hosting malwares or viruses, phishing and scam hosts, as shown in Table 12. Surprisingly, according to both VirusTotal and WOT'ratings, AURA found that 2,025 apps (15% of the dataset) talk to blacklisted domains. These blacklisted domains pose a wider rage of dangers to end-users, e.g., users' sensitive data could be leaked to these domains for illegal purposes, or users could end up downloading and installing malware. We provide a detailed break-down of blacklisted site categories in Table 13. As we can see, they include advertising services, hosting services, financial services, etc. We have manually browsed some

Table 12. Definitions and descriptions of blacklisted types of domains from WOT.

Blacklist type	**Description**
malware	Site is blacklisted for hosting malware
phishing	Site is blacklisted for hosting a phishing page
scam	Site is blacklisted for hosting a scam

Table 13. Categories of blacklisted domains and their percentages.

Category	**%**
Advertising	28
Hosting Service	23
Entertainment Media	14
Short URL Service	8
Dating	7
Social	7
DNS service	6
Online Shopping	3
Finance	2
Misc	2

of the blacklisted domains to discover how they lure users in and how they exploit users and there information. We provide details on their nefarious behavior next.

Luring users in. Blacklisted websites first use a "lure-in" to entice users into visiting the website or clicking on download links, namely, using these means:

(1) "Big reward" return trap: cheat users by claiming that they could obtain a big return after they buy the advertised promotions from the website.
(2) Adult content: using explicit images to lure users into subscribing to services.
(3) Intrusive ads, that is display ads that constantly pop up, counting on user's attrition to eventually click on the ad.
(4) Fake sites: present a deceiving front page (e.g., news, government, bank, travel) to lure users into sharing their sensitive and confidential information.
(5) Exploited sites: sites or hosting services that are compromised by malware.
(6) Abusing short URL and DNS services: using an URL shortener to hide their suspicious intentions and redirect the users to malicious sites.

Inflicting malicious behavior. Once users are lured into visiting websites, sharing information, or downloading software, the blacklisted websites exploit users' good faith by inflicting malicious behavior in a variety of ways, which we describe next.

Phishing sensitive personal information or confidential financial information, e.g., credit card details. Once users share confidential information, websites will resort to identity theft, credit card abuse, and tracking users' habits.

Distribution of rootkits, trojans, viruses, malware, spyware, rogues and adware, and creating virus attacks. Once such nefarious software is installed, the malicious behavior can take a variety of forms: corrupting the data saved on the smartphone, which could render the phone unusable, information leaking (e.g., financial information, passwords), and so on.

Intrusively and aggressively sell ads. Once such adware is installed, it displays non-stopping pop-up ads that users cannot dismiss/unsubscribe from.

Note that a blacklisted domain may have two or more of these behaviors, which means some of these intentions can co-exist.

5.1.4. *Potential uses and deployment:*

We envision using AURA in several different ways.

a. Advisory stand-alone tool. AURA could be used as an advisory stand-alone tool, where users submit the apps of interest, and receive an assessment; the set of users includes researchers that want to further study app security from an Internet access point of view.

b. Expanded app information. AURA could enhance the information presented to an user prior to installing an app. The Google Play market information panel could include AURA's assessment as a part of the profile of the app as a more refined explanation of the Internet Access permission.

c. App filtering. AURA could also be used as a filter before the app is allowed to enter Google Play. The market owner, such as Google, Samsung or Amazon, could force developers to evaluate their apps with AURA, and allow apps on the market only if they meet certain requirements (not talking to malware-hosting sites seems like a good requirement).

d. A component in a larger security system. AURA could be integrated into other static and dynamic analysis tools to provide more comprehensive risk information for each app. The interactions between the developer and market administrators are encouraged during the development and maintenance of the app.

6. Security: App Profiling

In this section we present an approach for profiling Android apps to understand the security implications of running those apps. For example, some apps use sensitive resources without disclosing the access up-front. Other apps can threaten availability due to excessive resource consumption, a condition that our approach detects.

A fundamental challenge for both users and app marketplaces such as Google Play or Apple App Store is how to understand and summarize app behavior. More specifically, in light of the 1,000,000+ apps currently on Google Play (ex Android Market),[2] we seek to answer the question "*Given an Android app, how can we efficiently get an informative thumbnail of its behavior?*" To this end have devised a profiling scheme that works even with limited resources in terms of time, manual effort, and cost. We define limited resources to mean: a few users with a few minutes of experimentation per application. At the same time, we want the resulting app profiles to be comprehensive, useful, and intuitive. Therefore, given an app and one or more short executions, we want a profile that captures succinctly what the app did, and contrast it with: (a) what it was expected or allowed to do, and (b) other executions of the same app. For example, an effective profile should provide: (a) how apps use resources, expressed in terms of network data and system calls, (b) the types of device resources (e.g., camera, telephony) an app accesses, and whether it is allowed to, and (c) what entities an app communicates with (e.g., cloud or third-party servers).

Who would be interested in such a capability? We argue that an inexpensive solution would appeal to everyone who "comes in contact" with the app, including: (a) the app developer, (b) the owner of an Android app market, (c) a system administrator, and (d) the end user. Effective profiling can help us: (a) enhance user control, (b) improve user experience, (c) assess performance and security implications, and (d) facilitate troubleshooting. We envision our quick and cost-effective thumbnails (profiles) to be the first step of app profiling, which can then have more involved and resource-intense steps, potentially based on what the thumbnail has revealed.

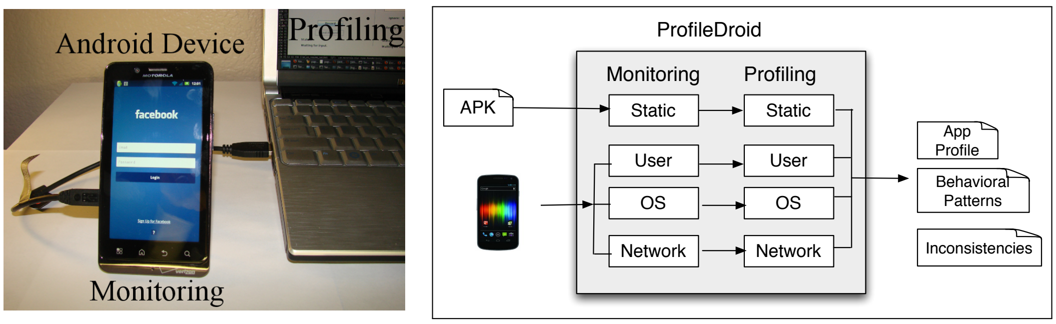

Fig. 12. Overview and actual usage (left) and architecture (right) of ProfileDroid.

Despite the flurry of research activity in this area, there is no approach yet that focuses on profiling the behavior of an Android app *itself* in all its complexity. Several efforts have focused on analyzing the mobile phone traffic and show the protocol related properties, but they do not study the apps *themselves*.[42,46] Others have studied security issues that reveal the abuse of personal device information.[60,90] However, all these works: (a) do not focus on *individual* apps, but report general trends, or (b) focus on a single layer, studying, e.g., the network behavior or the app specification in isolation. For example, some apps have negligible user inputs, such as `Pandora`, or negligible network traffic, such as `Advanced Task Killer`, and thus, by focusing only on one layer, the most significant aspect of an application could be missed.

Specifically, this section discusses the design and implementation of ProfileDroid,[94] a systematic and comprehensive system for profiling Android apps. A key novelty is that our profiling spans four layers: (a) static, i.e., app specification, (b) user interaction, (c) operating system, and (d) network. To the best of our knowledge, this is the first work[d] that considers all these layers in profiling individual Android apps. Our contributions are twofold. First, designing the system requires the careful selection of informative and intuitive metrics, which capture the essence of each layer. Second, implementing the system is a non-trivial task, and we have to overcome numerous practical challenges.[e]

For each layer, the monitoring component runs on the Android device where the app is running. The captured information is subsequently fed into the profiling part, which runs on the connected computer. In Fig. 12, on the right, we show a high level overview of our system and its design. On the left, we have an actual picture of the actual system the Android device that runs the app and the profiling computer (such as a desktop or a laptop).

[d]An earlier work[35] uses the term "cross-layer," but the layers it refers to are quite different from the layers we use.
[e] Examples include fine-tuning data collection tools to work on Android, distinguishing between presses and swipes, and disambiguating app traffic from third-party traffic.

From an architectural point of view, we have designed ProfileDroid to be flexible and modular with level-defined interfaces between the monitoring and profiling components. Thus, it is easy to modify or improve functionality within each layer. Furthermore, we could easily extend the current functionality to add more metrics, and even potentially more layers, such as a physical layer (temperature, battery level, etc.).

6.1. *Overview of Approach*

We present an overview of the design and implementation of ProfileDroid. We measure and profile apps at four different layers: (a) static, or app specification (b) user interaction, (c) operating system, and (d) network. For each layer, our system consists of two parts: a monitoring and a profiling component. For each layer, the monitoring component runs on the Android device where the app is running. The captured information is subsequently fed into the profiling part, which runs on the connected computer. In Fig. 12, on the right, we show a high level overview of our system and its design. On the left, we have an actual picture of the actual system the Android device that runs the app and the profiling computer (such as a desktop or a laptop).

In the future, we foresee a light-weight version of the whole profiling system to run exclusively on the Android device. The challenge is that the computation, the data storage, and the battery consumption must be minimized. How to implement the profiling in an incremental and online fashion is beyond the scope of the current work. Note that our system is focused on profiling of an *individual* app, and not intended to monitor user behavior on mobile devices.

From an architectural point of view, we design ProfileDroid to be flexible and modular with level-defined interfaces between the monitoring and profiling components. Thus, it is easy to modify or improve functionality within each layer. Furthermore, we could easily extend the current functionality to add more metrics, and even potentially more layers, such as a physical layer (temperature, battery level, etc.).

6.1.1. *Implementation and Challenges*

We describe the implementation of monitoring at each layer, and briefly touch on challenges we had to surmount when constructing ProfileDroid.

To profile an application, we start the monitoring infrastructure (described at length below) and then the target app is launched. The monitoring system logs all the relevant activities, e.g., user touchscreen input events, system calls, and all network traffic in both directions.

Static Layer. At the static layer, we analyze the APK (Android application package) file, which is how Android apps are distributed. We use `apktool` to unpack the APK file to extract relevant data. From there, we mainly focus on the

`Manifest.xml` file and the bytecode files contained in the `/smali` folder. The manifest is specified by the developer and identifies hardware usage and permissions requested by each app. The `smali` files contain the app bytecode which we parse and analyze statically, as explained later in Section 6.2.1.

User Layer. At the user layer, we focus on user-generated events, i.e., events that result from interaction between the user and the Android device while running the app. To gather the data of the user layer, we use a combination of the `logcat` and `getevent` tools of `adb`. From the `logcat` we capture the system debug output and log messages from the app. In particular, we focus on events-related messages. To collect the user input events, we use the `getevent` tool, which reads the `/dev /input/event*` to capture user events from input devices, e.g., touchscreen, accelerometer, proximity sensor. Due to the raw nature of the events logged, it was challenging to disambiguate between swipes and presses on the touchscreen. We provide details in Section 6.2.2.

Operating System Layer. At the operating system-layer, we measure the operating system activity by monitoring system calls. We collect system calls invoked by the app using an Android-specific version of `strace`. Next, we classify system calls into four categories: filesystem, network, VM/IPC, and miscellaneous. As described in Section 6.2.3, this classification is challenging, due to the virtual file system and the additional VM layer that decouples apps from the OS.

Network Layer. At the network layer, we analyze network traffic by logging the data packets. We use an Android-specific version of `tcpdump` that collects all network traffic on the device. We parse, domain-resolve, and classify traffic. As described in Section 6.2.4, classifying network traffic is a significant challenge in itself; we used information from domain resolvers, and improve its precision with manually-gathered data on specific websites that act as traffic sources.

6.1.2. *Experimental Setup*

Android Devices. The Android devices monitored and profiled in this paper were a pair of identical Motorola Droid Bionic phones, which have dual-core ARM Cortex-A9 processors running at 1GHz. The phones were released on released September 8, 2011 and run Android version 2.3.4 with Linux kernel version 2.6.35.

App Selection. To ensure representative results, we strictly follow the following criteria in selecting our test apps. First, we selected a variety of apps that cover most app categories as defined in Google Play, such as Entertainment, Productivity tools, etc. Second, all selected apps had to be popular, so that we could examine real-world, production-quality software with a broad user base. In particular, the selected apps must have at least 1,000,000 installs, as reported by Google Play, and be within the Top-130 free apps, as ranked by the Google Play website. In the end, we selected 27 apps as the basis for our study: 19 free apps and 8 paid apps; the 8 paid apps have free counterparts, which are included in the list of 19 free apps. The list of the selected apps, as well as their categories, is shown in Table 14.

Table 14. The test apps; app-$$ represents the paid version of an app.

App name	**Category**
Dictionary.com, Dictionary.com-$$	Reference
Tiny Flashlight	Tools
Zedge	Personalization
Weather Bug, Weather Bug-$$	Weather
Advanced Task Killer, Advanced Task Killer-$$	Productivity
Flixster	Entertainment
Picsay, Picsay-$$	Photography
ESPN	Sports
Gasbuddy	Travel
Pandora	Music & Audio
Shazam, Shazam-$$	Music & Audio
Youtube	Media & Video
Amazon	Shopping
Facebook	Social
Dolphin, Dolphin-$$	Communication (Browsers)
Angry Birds, Angry Birds-$$	Games
Craigslist	Business
CNN	News & Magazines
Instant Heart Rate, Instant Heart Rate-$$	Health & Fitness

Conducting the experiment. In order to isolate app behavior and improve precision when profiling an app, we do not allow other manufacturer-installed apps to run concurrently on the Android device, as they could interfere with our measurements. Also, to minimize the impact of poor wireless link quality on apps, we used WiFi in strong signal conditions. Further, to ensure statistics were collected of only the app in question, we installed one app on the phone at a time and uninstalled it before the next app was tested. Note however, that system daemons and required device apps were still able to run as they normally would, e.g., the service and battery managers.

Finally, in order to add stability to the experiment, the multi-layer traces for each individual app were collected from tests conducted by multiple users to obtain a comprehensive exploration of different usage scenarios of the target application. To cover a larger variety of running conditions without burdening the user, we use *capture-and-replay*, as explained below. Each user ran each app one time for 5 minutes; we capture the user interaction using event logging. Then, using a replay tool we created, each recorded run was replayed back 5 times in the morning and 5 times at night, for a total of 10 runs each per user per app. The runs of each app were conducted at different times of the day to avoid time-of-day bias, which could lead to uncharacteristic interaction with the app; by using the capture-and-replay tool, we are able to achieve this while avoiding repetitive manual runs from the same user. For those apps that had both free and paid versions, users carried out the same task, so we can pinpoint differences between paid and free versions. To summarize, our profiling is based on 30 runs (3 users $\times$ 10 replay runs) for each app.

6.2. *Analyzing each layer*

In this section, we first provide detailed descriptions of our profiling methodology, and we highlight challenges and interesting observations.

6.2.1. *Static Layer*

The first layer in our framework aims at understanding the app's functionality and permissions. In particular, we analyze the APK file on two dimensions to identify app functionality and usage of device resources: first, we extract the permissions that the app asks for, and then we parse the app bytecode to identify intents, i.e., indirect resource access via deputy apps. Note that, in this layer only, we analyze the app without running it — hence the name *static layer.*

Functionality usage. Android devices offer several major functionalities, labeled as follows: Internet, GPS, Camera, Microphone, Bluetooth and Telephony. We present the results in Table 15. A '✓' means the app requires permission to use the device, while 'I' means the device is used indirectly via intents and deputy apps. We observe that Internet is the most-used functionality, as the Internet is the gateway to interact with remote servers via 3G or WiFi — all of our examined apps use the Internet for various tasks. For instance, `Pandora` and `YouTube` use the Internet to fetch multimedia files, while `Craigslist` and `Facebook` use it to get content updates when necessary.

GPS, the second most popular resource (9 apps) is used for navigation and location-aware services. For example, `Gasbuddy` returns gas stations near the user's location, while `Facebook` uses the GPS service to allow users to *check-in*, i.e., publish their presence at entertainment spots or places of interests. Camera, the third-most popular functionality (5 apps) is used for example, to record and post real-time news information (`CNN`), or for for barcode scanning `Amazon`. Microphone, Bluetooth and Telephony are three additional communication channels besides the Internet, which could be used for voice communication, file sharing, and text messages. This increased usage of various communication channels is a double-edged sword. On the one hand, various communication channels improve user experience. On the other hand, it increases the risk of privacy leaks and security attacks on the device.

Intent usage. Android intents allow apps to access resources indirectly by using deputy apps that have access to the requested resource. For example, `Facebook` does not have the camera permission, but can send an intent to a deputy camera app to take and retrieve a picture.[f] We decompiled each app using `apktool` and identified instances of the `android.content.Intent` class in the Dalvik bytecode. Next, we analyzed the parameters of each intent call to find the intent's type, i.e., the device's resource to be accessed via deputy apps.

We believe that presenting users with the list of resources used via intents (e.g., that the `Facebook` app does not have direct access to the camera, but nevertheless it can use the camera app to take pictures) helps them make better-informed decisions about installing and using an app. Though legitimate within the Android security model, this lack of user forewarning can be considered deceiving; with the more comprehensive picture provided by ProfileDroid, users have a better understanding of resource usage, direct or indirect.[12]

6.2.2. *User Layer*

At the user layer, we analyze the input events that result from user interaction. In particular, we focus on *touches* — generated when the user touches the screen — as touchscreens are the main Android input devices. Touch events include *presses*, e.g., pressing the app buttons of the apps, and *swipes* — finger motion without losing contact with the screen. The intensity of events (events per unit of time), as well as the ratio between swipes and presses are powerful metrics for GUI behavioral fingerprinting (Section 6.3.4); we present the results in Fig. 13 and now proceed to discussing these metrics.

Technical challenge. Disambiguating between swipes and presses was a challenge, because of the nature of reported events by the *getevent* tool. Swipes and presses are reported by the touchscreen input device, but the reported events are not labeled as swipes or presses. A single press usually accounts for 30 touchscreen events, while a swipe usually accounts for around 100 touchscreen events. In order to distinguish between swipes and presses, we developed a method to cluster and label events. For example, two events separated by less than 80 milliseconds are likely to be part of a sequence of events, and if that sequence of events grows above 30, then it is likely that the action is a swipe instead of a press. Evaluating and fine-tuning our method was an intricate process.

[f]This was the case for the version of the Facebook app we analyzed in March 2012, the time we performed the study. However, we found that, as of June 2012, the Facebook app requests the Camera permission explicitly.

Table 15. Profiling results of *static* layer; '✓' represents use via permissions, while 'I' via intents.

App	**Internet**	**GPS**	**Camera**	**Microphne**	**Bluetooth**	**Telephony**
Dictionary.com	✓			I		I
Dictionary.com-$$	✓			I		I
Tiny Flashlight	✓		✓			
Zedge	✓					
Weather Bug	✓	✓				
Weather Bug-$$	✓	✓				
Advanced Task Killer	✓					
Advanced Task Killer-$$	✓					
Flixster	✓	✓				
Picsay	✓					
Picsay-$$	✓					
ESPN	✓					
Gasbuddy	✓	✓				
Pandora	✓				✓	
Shazam	✓	✓		✓		
Shazam-$$	✓	✓		✓		
YouTube	✓					
Amazon	✓		✓			
Facebook	✓	✓	I			✓
Dolphin	✓	✓				
Dolphin-$$	✓	✓				
Angry Birds	✓					
Angry Birds-$$	✓					
Craigslist	✓					
CNN	✓		✓			
Instant Heart Rate	✓		✓		I	I
Instant Heart Rate-$$	✓		✓		I	I

Touch events intensity. We measured touch intensity as the number of touch events per second — this reveals how interactive an app is. For example, the music app `Pandora` requires only minimal input (music control) once a station is selected. In contrast, in the game `Angry Birds`, the user has to interact with the interface of the game using swipes and screen taps, which results in a high intensity for touch events.

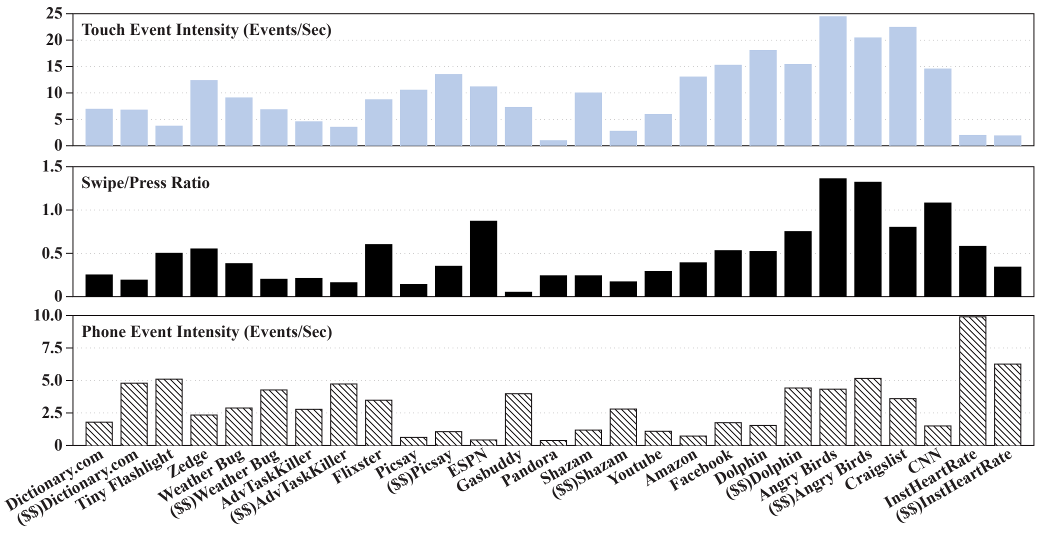

Fig. 13. Profiling results of *user* layer; note that scales are different.

Swipe/Press ratio. We use the ratio of swipes to presses to better capture the nature of the interaction, and distinguish between apps that have similar touch intensity. Note that swipes are used for navigation and zooming, while touches are used for selection. Figure 13 shows that apps that involve browsing, news-page flipping, gaming, e.g., `CNN`, `Angry Birds`, have a high ratio of swipes to presses; even for apps with the same touch intensity, the swipe/press ratio can help profile and distinguish apps, as seen in the following table:

App	Touch intensity	Swipe/Press ratio
`Picsay`	*medium*	*low*
`CNN`	*medium*	*high*

Phone event intensity. The bottom chart in Fig. 13 shows the intensity of events generated by the phone itself during the test. These events contain a wealth of contextual data that, if leaked, could pose serious privacy risks. The most frequent events we observed were generated by the accelerometer, the light proximity sensor, and for some location-aware apps, the compass. For brevity, we omit details, but we note that phone-event intensity, and changes in intensity, can reveal the user's proximity to the phone, the user's motion patterns, and user orientation and changes thereof.

6.2.3. *Operating System Layer*

We first present a brief overview of the Android OS, and then discuss metrics and results at the operating system layer.

Android OS is a Linux-based operating system, customized for mobile devices. Android apps are written in Java and compiled to Dalvik executable (Dex) bytecode. The bytecode is bundled with the app manifest (specification, permissions) to create an APK file. When an app is installed, the user must grant the app the permissions specified in the manifest. The Dex bytecode runs on top of the Dalvik Virtual Machine (VM) — an Android-specific Java virtual machine. Each app runs as a separate Linux process with a unique user ID in a separate copy of the VM. The separation among apps offers a certain level of protection and running on top of a VM avoids granting apps direct access to hardware resources. While increasing reliability and reducing the potential for security breaches, this vertical (app–hardware) and horizontal (app–app) separation means that apps do not run natively and inter-app communications must take place primarily via IPC. We profile apps at the operating system layer with several goals in mind: to understand how apps use system resources, how the operating-system intensity compares to the intensity observed at other layers, and to characterize the potential performance implications of running apps in separate VM copies. To this end, we analyzed the system call traces for each app to understand the nature and frequency of system calls. We present the results in Table 16.

System call intensity. The second column of Table 16 shows the system call intensity in system calls per second. While the intensity differs across apps, note that in all cases the intensity is relatively high (between 30 and 1,183 system calls per second) for a mobile platform.

System call characterization. To characterize the nature of system calls, we group them into four bins: file system (FS), network (NET), virtual machine (VM&IPC), and miscellaneous (MISC). Categorizing system calls is not trivial.
Technical challenge. The Linux version running on our phone (2.6.35.7 for Arm) supports about 370 system calls; we observed 49 different system calls in our traces. While some system calls are straightforward to categorize, the operation of virtual filesystem calls such as `read` and `write`, which act on a file descriptor, depends on the file descriptor and can represent file reading and writing, network send/receive, or reading/altering system configuration via `/proc`. Therefore, for all the virtual filesystem calls, we categorize them based on the file descriptor associated with them, as explained below. FS system calls are used to access data stored on the flash drive and SD card of the mobile device and consist mostly of `read` and `write` calls on a file descriptor associated with a space-occupying file in the file system, i.e., opened via `open`. NET system calls consist mostly of `read` and `write` calls on a file descriptor associated with a network socket, i.e., opened via `socket`; note that for NET system calls, reads and writes mean receiving from and sending to the network. VM&IPC system calls are calls inserted by the virtual machine for operations such as scheduling, timing, idling, and IPC. For each such operation, the VM inserts a specific sequence of system calls. We extracted these sequences, and compared the

Table 16. Profiling results: *operating system* layer.

App	**Syscall intensity (calls/sec.)**	**FS (%)**	**NET (%)**	**VM&IPC (%)**	**MISC (%)**
Dictionary.com	1025.64	3.54	1.88	67.52	27.06
Dictionary.com-$$	492.90	7.81	4.91	69.48	17.80
Tiny Flashlight	435.61	1.23	0.32	77.30	21.15
Zedge	668.46	4.17	2.25	75.54	18.04
Weather Bug	1728.13	2.19	0.98	67.94	28.89
Weather Bug-$$	492.17	1.07	1.78	75.58	21.57
AdvTaskKiller	75.06	3.30	0.01	65.95	30.74
AdvTaskKiller-$$	30.46	7.19	0.00	63.77	29.04
Flixster	325.34	2.66	3.20	71.37	22.77
Picsay	319.45	2.06	0.01	75.12	22.81
Picsay-$$	346.93	2.43	0.16	74.37	23.04
ESPN	1030.16	2.49	2.07	87.09	8.35
Gasbuddy	1216.74	1.12	0.32	74.48	24.08
Pandora	286.67	2.92	2.25	70.31	24.52
Shazam	769.54	6.44	2.64	72.16	18.76
Shazam-$$	525.47	6.28	1.40	74.31	18.01
YouTube	246.78	0.80	0.58	77.90	20.72
Amazon	692.83	0.42	6.33	76.80	16.45
Facebook	1030.74	3.99	2.98	72.02	21.01
Dolphin	850.94	5.20	1.70	71.91	21.19
Dolphin-$$	605.63	9.05	3.44	68.45	19.07
Angry Birds	1047.19	0.74	0.36	82.21	16.69
Angry Birds-$$	741.28	0.14	0.04	85.60	14.22
Craigslist	827.86	5.00	2.47	73.81	18.72
CNN	418.26	7.68	5.55	71.47	15.30
InstHeartRate	944.27	7.70	1.73	75.48	15.09
InstHeartRate-$$	919.18	12.25	0.14	72.52	15.09

number of system calls that appear as part of the sequence to the total number, to quantify the VM and IPC-introduced overhead. The most common VM/IPC system calls we observed (in decreasing order of frequency) were: `clock_gettime`, `epoll_wait`, `getpid`, `getuid32`, `futex`, `ioctl`, and `ARM_cacheflush`. The remaining system calls are predominantly `read` and `write` calls to the `/proc` special filesystem are categorized as MISC.

The results are presented in Table 16: for each category, we show both intensity, as well as the percentage relative to all categories. Note that FS and NET percentages are quite similar, but I/O system calls (FS and NET) constitute a relatively small percentage of total system calls, with the VM&IPC dominating. We will come back to this aspect in Section 6.3.

6.2.4. *Network Layer*

The network-layer analysis summarizes the data communication of the app via WiFi or 3G. Android apps increasingly rely on Internet access for a diverse array of

Table 17. Profiling results of *network* layer; '–' represents no traffic.

App	**Traffic intens. (bytes/ sec.)**	**Traffic In/Out (ratio)**	**Origin (%)**	**CDN+ Cloud (%)**	**Google (%)**	**Third party (%)**	**Traffic srcs.**	**HTTP/ HTTPS split (%)**
Dictionary.com	1450	1.94	–	35.36	64.64	–	8	100/–
Dictionary.com-$$	488	1.97	0.02	1.78	98.20	–	3	100/–
Tiny Flashlight	134	2.49	–	–	99.79	0.21	4	100/–
Zedge	15424	10.68	–	96.84	3.16	–	4	100/–
Weather Bug	3808	5.05	–	75.82	16.12	8.06	13	100/–
Weather Bug-$$	2420	8.28	–	82.77	6.13	11.10	5	100/–
AdvTaskKiller	25	0.94	–	–	100.00	–	1	91.96/8.04
AdvTaskKiller-$$	–	–	–	–	–	–	0	–/–
Flixster	23507	20.60	2.34	96.90	0.54	0.22	10	100/–
Picsay	4	0.34	–	48.93	51.07	–	2	100/–
Picsay-$$	320	11.80	–	99.85	0.15	–	2	100/–
ESPN	4120	4.65	–	47.96	10.09	41.95	5	100/–
Gasbuddy	5504	10.44	6.17	11.23	81.37	1.23	6	100/–
Pandora	24393	28.07	97.56	0.91	1.51	0.02	11	99.85/0.15
Shazam	4091	3.71	32.77	38.12	15.77	3.34	13	100/–
Shazam-$$	1506	3.09	44.60	55.36	0.04	–	4	100/–
YouTube	109655	34.44	96.47	–	3.53	–	2	100/–
Amazon	7757	8.17	95.02	4.98	–	–	4	99.34/0.66
Facebook	4606	1.45	67.55	32.45	–	–	3	22.74/77.26
Dolphin	7486	5.92	44.55	0.05	8.60	46.80	22	99.86/0.14
Dolphin-$$	3692	6.05	80.30	1.10	5.80	12.80	9	99.89/0.11
Angry Birds	501	0.78	–	73.31	10.61	16.08	8	100/–
Angry Birds-$$	36	1.10	–	88.72	5.79	5.49	4	100/–
Craigslist	7657	9.64	99.97	–	–	0.03	10	100/–
CNN	2992	5.66	65.25	34.75	–	–	2	100/–
InstHeartRate	573	2.29	–	4.18	85.97	9.85	3	86.27/13.73
InstHeartRate-$$	6	0.31	–	8.82	90.00	1.18	2	20.11/79.89

services, e.g., for traffic, map or weather data and even offloading computation to the cloud. An increasing number of network traffic sources are becoming visible in app traffic, e.g., Content Distribution Networks, Cloud, Analytics and Advertisement. To this end, we characterize the app's network behavior using the following metrics and present the results in Table 17.

Traffic intensity. This metric captures the intensity of the network traffic of the app. Depending on the app, the network traffic intensity can vary greatly, as shown in Table 17. For the user, this great variance in traffic intensity could be an important property to be aware of, especially if the user has a limited data plan. Not surprisingly, we observe that the highest traffic intensity is associated with a video app, `YouTube`. Similarly, the entertainment app `Flixster`, music app `Pandora`, and personalization app `Zedge` also have large traffic intensities as they download audio and video files. We also observe apps with zero, or negligible, traffic intensity, such as the productivity app `Advanced Task Killer` and free photography app `Picsay`.

Origin of traffic. The origin of traffic means the percentage of the network traffic that comes from the servers owned by the app provider. This metric is particularly interesting for privacy-sensitive users, since it is an indication of the control that the app provider has over the app's data. Interestingly, there is large variance for this metric, as shown in Table 17. For example, the apps `Amazon`, `Pandora`, `YouTube`, and `Craigslist` deliver most of their network traffic (e.g., more than 95%) through their own servers and network. However, there is no origin traffic in the apps `Angry Birds` and `ESPN`. Interestingly, we observe that ony 67% of the `Facebook` traffic comes from Facebook servers, with the remaining coming from content providers or the cloud.

Technical challenge. It is a challenge to classify the network traffic into different categories (e.g., cloud vs. ad network), let alone identify the originating entity. To resolve this, we combine an array of methods, including reverse IP address lookup, DNS and `whois`, and additional information and knowledge from public databases and the web. In many cases, we use information from CrunchBase (crunchbase.com) to identify the type of traffic sources after we resolve the top-level domains of the network traffic.[17] Then, we classify the remaining traffic sources based on information gleaned from their website and search results.

In some cases, detecting the origin is even more complicated. For example, consider the `Dolphin` web browser — here the origin is not the Dolphin web site, but rather the website that the user visits with the browser, e.g., if the user visits CNN, then cnn.com is the origin. Also, YouTube is owned by Google and YouTube media content is delivered from domain 1e100.net, which is owned by Google; we report the media content (96.47%) as Origin, and the remaining traffic (3.53%) as Google which can include Google ads and analytics.

CDN+Cloud traffic. This metric shows the percentage of the traffic that comes from servers of CDN (e.g., Akamai) or cloud providers (e.g., Amazon AWS). Content Distribution Network (CDN) has become a common method to distribute the app's data to its users across the world faster, with scalability and cost-effectively. Cloud platforms have extended this idea by providing services (e.g., computation) and not just data storage. Given that it is not obvious if someone using a cloud service is using it as storage, e.g., as a CDN, or for computation, we group CDN and cloud services into one category. Interestingly, there is a very strong presence of this kind of traffic for some apps, as seen in Table 17. For example, the personalization app `Zedge`, and the video-heavy app `Flixster` need intensive network services, and they use CDN and Cloud data sources. The high percentages that we observe for CDN+Cloud traffic point to how important CDN and Cloud sources are, and how much apps rely on them for data distribution.

Google traffic. Given that Android is a product of Google, it is natural to wonder how involved Google is in Android traffic. The metric is the percentage of traffic exchanged with Google servers (e.g., 1e100.net), shown as the second-to-last column in Table 17. It has been reported that the percentage of Google traffic has increased significantly over the past several years.[23] This is due in part to the increasing penetration of Google services (e.g., maps, ads, analytics, and Google App Engine). Note that 22 of out of the 27 apps exchange traffic with Google, and we discuss this in more detail in Section 6.3.

Third-party traffic. This metric is of particular interest to privacy-sensitive users. We define third party traffic as network traffic from various advertising services (e.g., Atdmt) and analytical services (e.g., Omniture) besides Google, since advertising and analytical services from Google are included in the Google traffic metric. From Table 17, we see that different apps have different percentages of third-party traffic. Most apps only get a small or negligible amount of traffic from third parties (e.g., `YouTube`, `Amazon` and `Facebook`). At the same time, nearly half of the total traffic of `ESPN` and `Dolphin` comes from third parties.

The ratio of incoming traffic and outgoing traffic. This metric captures the role of an app as a consumer or producer of data. In Table 17, we see that most of the apps are more likely to receive data than to send data. As expected, we see that the network traffic from `Flixster`, `Pandora`, and `YouTube`, which includes audio and video content, is mostly incoming traffic as the large values of the ratios show. In contrast, apps such as `Picsay` and `Angry Birds` tend to send out more data than they receive.

Note that this metric could have important implications for performance optimization of wireless data network providers. An increase in the outgoing traffic could challenge network provisioning, in the same way that the emergence of p2p file sharing stretched cable network operators, who were not expecting large household upload needs. Another use of this metric is to detect suspicious variations in the ratio, e.g., unusually large uploads, which could indicate a massive theft of data. Note that the goal of this paper is to provide the framework and tools for such an investigation, which we plan to conduct as our future work.

Number of distinct traffic sources. An additional way of quantifying the interactions of an app is with the number of distinct traffic sources, i.e., distinct top-level domains. This metric can be seen as a complementary way to quantify network interactions, a sudden increase in this metric could indicate malicious behavior. In Table 17 we present the results. First, we observe that all the examined apps interact with at least two distinct traffic sources, except `Advanced Task Killer`. Second, some of the apps interact with a surprisingly high number of distinct traffic sources, e.g., `Weather bug`, `Flixster`, and `Pandora`. Note that we count all the distinct traffic sources that appear in the traces of multiple executions.

The percentage of HTTP and HTTPS traffic. To get a sense of the percentage of secure Android app traffic, we compute the split between HTTP and HTTPS traffic, e.g., non-encrypted and encrypted traffic. We present the results in the last column of Table 17 ('–' represents no traffic). The absence of HTTPS traffic is staggering in the apps we tested, and even `Facebook` has roughly 22 % of unencrypted traffic, as we further elaborate in Section 6.3.

6.3. *Interpreting the Results*

6.3.1. *Privacy and Security Issues*

Lack of transparency. We identify discrepancies between the app specification and app execution. For example, `Instant Heart Rate` and `Dictionary` use telephony resources without declaring them up-front, via intents (by asking a proxy app).

Most network traffic is unencrypted. We find that most of the network traffic is not encrypted. For example, most of the web-based traffic is over HTTP and not HTTPS: only 8 out of the 27 apps use HTTPS and for `Facebook`, 22.74% of the traffic is not encrypted.

6.3.2. *Operational Issues*

Free apps have a cost. Free versions of apps could end up costing more than their paid versions especially on limited data plans, due to increased advertising/analytics traffic. For example, the free version of `Angry Birds` has 13 times more traffic than the paid version.

Apps talk to "strangers". Apps interact with many more traffic sources than one would expect. For example, the free version of `Shazam` talks to 13 different traffic sources in a 5-minute interval, while its paid counterpart talks with 4.

Google "touches" almost everything. Out of 27 apps, 22 apps exchange data traffic with Google, including apps that one would not have expected, e.g., Google accounts for 85.97% of the traffic for the free version of the health app `Instant Heart Rate`, and 90% for the paid version.

6.3.3. *Performance Issues*

Security comes at a price. The Android OS uses virtual machine (VM)-based isolation for security and reliability, but as a consequence, the VM overhead is high: more than 63% of the system calls are introduced by the VM for context-switching between threads, supporting IPC, and idling.[94]

6.3.4. *Thumbnails*

The intensity of activities at each layer is a fundamental metric that we want to capture, as it can provide a thumbnail of the app behavior. The multi-layer intensity is a tuple consisting of intensity metrics from each layer: static (number of functionalities), user (touch event intensity), operating system (system call intensity), and network (traffic intensity).

Presenting raw intensity numbers is easy, but it has limited intuitive value. For example, reporting 100 system calls per second provides minimal information to a user or an application developer. A more informative approach is to present the relative intensity of this app compared to other apps.

We opt to represent the activity intensity of each layer using labels: H (high), M (medium), and L (low). The three levels (H, M, L) are defined relative to the intensities observed at each layer using the five-number summary from statistical analysis:[31] minimum (Min), lower quartile (Q_1), median (Med), upper quartile (Q_3), and maximum (Max).

Table 18 shows the results of applying this H-M-L model to our test apps. We now proceed to showing how users and developers can benefit from an H-M-L-based app thumbnail for characterizing app behavior. Users can make more informed decisions when choosing apps by matching the H-M-L thumbnail with individual preference and constraints. For example, if a user has a small-allotment data plan on the phone, perhaps he would like to only use apps that are rated L for the intensity of network traffic; if the battery is low, perhaps she should refrain from running apps rated H at the OS or network layers.

Developers can also benefit from the H-M-L model by being able to profile their apps with ProfileDroid and optimize based on the H-M-L outcome. For example, if ProfileDroid indicates an unusually high intensity of filesystem calls

Table 18. Thumbnails of multi-layer intensity in the *H*-*M*-*L* model (*H*:high, *M*:medium, *L*:low).

App	**Static (# of func.)**	**User (events/ sec.)**	**OS (syscall/ sec.)**	**Network (bytes/ sec.)**
Dictionary.com	**L**	M	**H**	M
Dictionary.com-$$	**L**	M	M	M
Tiny Flashlight	M	**L**	M	**L**
Zedge	**L**	M	M	**H**
Weather Bug	M	M	**H**	M
Weather Bug-$$	M	M	M	M
AdvTaskKiller	**L**	M	**L**	**L**
AdvTaskKiller-$$	**L**	M	**L**	**L**
Flixster	M	M	**L**	**H**
Picsay	**L**	M	**L**	**L**
Picsay-$$	**L**	M	M	M
ESPN	**L**	M	**H**	M
Gasbuddy	M	M	**H**	M
Pandora	M	**L**	**L**	**H**
Shazam	**H**	**L**	M	M
Shazam-$$	**H**	**L**	**H**	M
YouTube	**L**	M	M	**H**
Amazon	M	M	M	**H**
Facebook	**H**	**H**	**H**	M
Dolphin	M	**H**	M	**H**
Dolphin-$$	M	**H**	M	M
Angry Birds	**L**	**H**	M	M
Angry Birds-$$	**L**	**H**	**H**	**L**
Craigslist	**L**	**H**	**H**	**H**
CNN	M	M	M	M
InstHeartRate	M	**L**	**H**	M
InstHeartRate-$$	M	**L**	**H**	**L**

in the operating system layer, the developer can examine their code to ensure those calls are legitimate. Similarly, if the developer is contemplating using an advertising library in their app, she can construct two *H*-*M*-*L* app models, with and without the ad library and understand the trade-offs.

In addition, an *H*-*M*-*L* thumbnail can help capture the nature of an app. Intuitively, we would expect interactive apps (social apps, news apps, games, Web browsers) to have intensity *H* at the user layer; similarly, we would expect media player apps to have intensity *H* at the network layer, but *L* at the user layer. Table 18 supports these expectations, and suggests that the the *H*-*M*-*L* thumbnail could be an initial way to classify apps into coarse behavioral categories.

Table 19. Android restart levels.

Level	**Cause**
1: Pause activity	Activity becomes (partially) covered; Turn off screen
2: Stop activity	Switch to another app; Start a new activity in the same app; Receive a phone call; Press ‘Home’ button
3: Destroy activity	Press ‘Back’ button; Kill app

7. Security: Moving Target Defense

In this section we present a Moving-target defense approach that aims to increase security by making the smartphone’s integrity and confidentiality harder to subvert.

Moving-target defense is an effective strategy for deflecting cyber attacks. The widespread use of smartphones in the tactical field requires novel ways of securing smartphones against an ever-increasing number of zero-day attacks. We have proposed[81] a new, proactive approach for securing smartphone apps against certain classes of attacks. Specifically, we have leveraged the smartphone’s native support for quick and lossless restarts to make application restart a cyber maneuver meant to deflect and confuse attackers. We have proposed a time-series entropy metric to quantify attack resilience. We have applied our approach to 12 popular Android apps chosen from a variety of domains, including online banking and shopping. Preliminary experiments with using proactive restarts on these apps show that restart is a promising way of increasing attack resilience for a certain class of side-channel attacks named Activity Inference attacks.

7.1. *Background*

We now present background information on proactive security, as well as the resume/restart mechanism on Android.

7.1.1. *Android Restart*

The Android smartphone platform consists of apps, usually written in Java, running on top of Dalvik, a Java virtual machine, which in turn runs on top of a smartphone-specific Linux kernel. Android apps, due to the nature of the platform, are centered around a GUI; an app’s GUI consists of separate “Activities”, where an activity roughly corresponds to a screen in a desktop program’s GUI. As a result of user interaction or outside events, an app transitions among activities; for example, in the Newegg online shopping app, if the user is in the Main activity and clicks the ‘My Account’ menu item, the app transitions to the Login activity (see Fig. 14).

Smartphones (unlike desktop or server systems) have limited resources. When the system is low on memory, or the user turns the screen off, or switches to a different app, the current app is automatically paused or even killed; a small percentage of

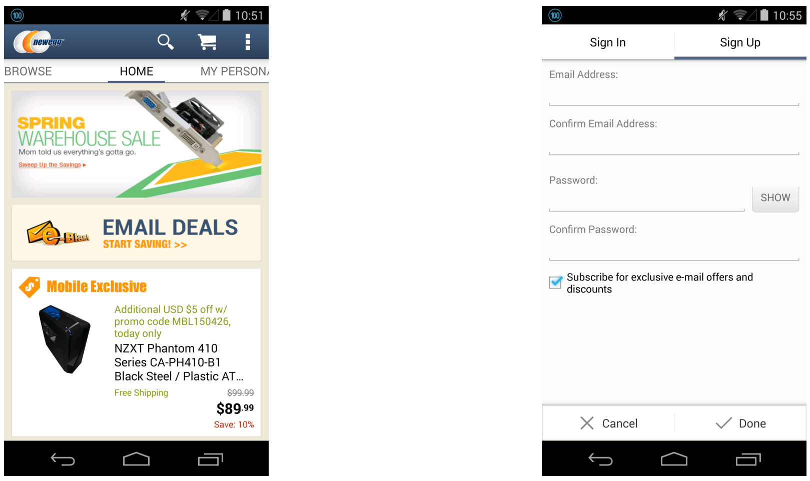

Fig. 14. Source activity (left) and destination activity (right).

apps that provide background services remain running, albeit in a restricted mode. When the user returns to the app, the app is resumed or restarted. Hence smartphone apps and OSs are designed from the ground up to support pause/resume operations smoothly and efficiently.

In Android, our target platform, there are three main levels of restart. We present these levels in Table 19: the level is on the left, and the cause is on the right. A restart cycle has little impact on the app and app state: the OS automatically saves and restores GUI state. However, at the OS level, a restart cycle (especially at level 3 — destroy app) is very disruptive, as the process is killed. When the app restarts, it restarts with different OS state, e.g., process identifier (PID), memory mapping, process counters from `/proc` files, etc.

Since restart is such a common and efficient operation on smartphones, and is gracefully tolerated by apps while being disruptive for the OS, our key insight is to use proactive restarts to change the attack surface hence offering a cyber-maneuver capability.

7.1.2. *Activity Inference Attacks*

Activity inference[26] represents a class of side-channel attacks where a malicious background application M can stealthily infer an activity transition occurring in a foreground benign app B. Further, M can precisely pinpoint which activity B is transitioning into, in real time. The attack is strong as it does not require any special permission. In fact, there is no vulnerability really being exploited, since all the information gathered by the malware M is publicly-available information including `/proc` files, e.g., `/proc/[pid]/statm`.

The fundamental weakness, exploited by such attacks, is that the information exposed through such channels happens to correlate well with B's activity transition behaviors. For instance, when an activity transition occurs in the foreground, the application process allocates screen buffer for the new activity as shared memory with a fixed size (proportional to the screen size) and then deallocates the buffer of the previous activity. Such unique memory consumption patterns can be easily captured through the `/proc` side channel. Furthermore, each destination activity has a different initial behavior, e.g., some activity's onCreate() callback may load an advertisement and therefore causes a new network connection to be created. Through other side channels, such initial behaviors are characterized to distinguish the destination activity.

The Activity inference attack has many consequences, one of which is that once the background malware M infers which foreground activity B is transitioning into, it can inject a phishing activity into the foreground to preempt B. The user will then be fooled into interacting with the malware M instead of the original app B.

Our scheme aims to address this fundamental weakness by using proactive restart to produce changes in OS state that are harder to predict, hence undermining the attacker's assumption that the side channel is reliable.

7.2. *Example*

We now present an example that motivates, as well as illustrates, our approach. Consider the Newegg Mobile app. An attacker might use an Activity Inference attack to try to determine which activity Newegg Mobile is in, and which activity it is transitioning to, so that the attacker can inject their own fake activity to try to phish secrets.

Let us suppose that Newegg Mobile is in the Main activity (Fig. 14 left) and is preparing to transition to the Login activity (Fig. 14 right). An Activity Inference attack relies on observing side-channel information, i.e., shared memory values in `/proc/pid/statm`, which will reveal a time series event. If the attacker detects this event quickly, then the attacker can "pop up" a fake activity that looks very similar to Login, and trick the user into inputting data into the fake activity — if this input data is sensitive information, such as a username/password combination (as is the case here), a credit card number, or a bank account number, the attack succeeds.

However, with our approach which injects restart events, the time series of shared memory values is confusing for the attacker: due to the perturbation introduced by restart, depending on where we choose to restart, there can be multiple time series with multiple events which represent strategies **S3** and **S4** defined in Section 7.3. In fact, our approach can deliberately insert restart events for the current activity just to confuse the attacker into believing there is an activity transition going on, when in fact there is no such transition.

Hence our proactive approach confuses the attacker into not knowing if, and when, the app is transitioning between activities.

7.3. *Implementation*

We now describe our testbed and implementation.

Environment. The smartphone we used for experiments was an LG Nexus 5 running Android version 4.4.4, Linux kernel version 3.4.0, on a four-core ARMv7 CPU@2.2 GHz.

Restart implementation. In Fig. 15 we show our implementation. In Android, applications use the services of the Android Framework (AF) and run on top of the Dalvik virtual machine, which in turn runs on top of a Linux kernel. The AF has a component named Activity Manager (AM) which is in charge of orchestrating app execution, including transition between activities. For simplicity, we only depict one running app, but in practice Android runs multiple apps concurrently. Let us assume that the app contains two activities, `A` and `B`, and due to an input event, e.g., the user pressing a button, the app wants to transition from `A` to `B`. In the standard implementation of Android, the activity transition will follow the "old pathway" (shown in gray color on top), that is, transition directly from `A` to `B`. In our implementation, the transition follows new pathways (shown in blue color) where there is an intervening restart, e.g., restart `A` prior to the transition, or restart `B` after the transition. Moreover, our approach supports a third new pathway where `A` is restarted even when no transition is necessary, to confuse the attacker.

We achieve this by using AM services: we use Android's `adb` shell to send messages to the AM, so that activity transitions follow the new pathways. We have used

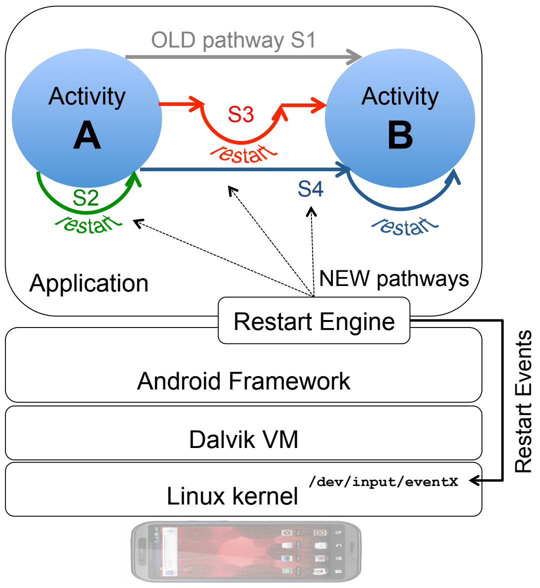

Fig. 15. Overview of our implementation.

restart level 2 (Section 7.1.1), that is, stop and restart the activity. Extending the approach to use restart levels 1 or 3 is straightforward.

Restart strategy. We experimented with four restart strategies, labeled **S1**–**S4**, that govern how the system should proceed when transitioning from activity A to activity B:

- **S1**: The "old" approach, without restart, where we just transition from activity A to activity B.
- **S2**: A restart approach without transition (just restart A).
- **S3**: Our main proposed restart approach: restart A, then transition from A to B.
- **S4**: An alternative restart approach: transition from A to B then restart B.

7.4. *Evaluation*

We now present our evaluation; first, we provide an overview of the apps and app selection process, then we discuss the experimental methodology and the results.

7.4.1. *Examined apps*

For evaluation we chose 24 activity transitions in 12 Android apps. We used several criteria when selecting the apps to ensure a representative sample: apps had to be popular, spanning free and paid categories; third-party and built-in categories; and have a wide range of sizes. In Table 20 we present the apps: name, popularity (number of installs per Google Play) and size. As we can see, 9 apps are free and 2 are paid (indicated by the $$ sign). Of the 9 free apps, 5 are third-party apps available on Google Play and 4 are built-in apps that come preinstalled with the phone. The built-in apps are particularly valuable and need to be protected for two reasons: (1) since they come from a trusted source, the vendor, they have higher privilege than third-party apps hence exploiting a vulnerability in such an app can inflict significant damage; and (2) preinstalled apps cannot be easily removed by regular users, since the phone needs to be "rooted" for the app to be removed.[92] Apps have a range of sizes, from medium (367 KB) to large (10 MB). Four of the third-party apps are very popular, having in excess of 1 million installs. Moreover, two of them — Chase Mobile and Newegg Mobile — are security-critical since they are used for online banking and online shopping; a security attack against them can expose the user's bank account information or credit card numbers.

7.4.2. *Data collection*

The data collection process is shown in Fig. 16. The test phone is connected to a laptop via the Android Debugging Bridge (ADB). We triggered restarts using ADB shell commands issued from the laptop. We wrote a monitor process — a native Linux process, rather than VM-based app — to monitor app execution, taking a sample every 8 milliseconds and collecting side-channel information. In particular,

Table 20. Test apps characteristics.

App	Popularity (# installs)	Size (KB)
Chase Mobile	10,000,000+	10,000
Newegg Mobile	1,000,000+	9,900
Browser	(builtin)	2,536
GBC Emulator ($$)	10,000+	367
OI File Manager	5,000,000+	973
Gallery3d	(builtin)	5,122
VideoEditor	(builtin)	5,243
Calendar	(builtin)	1,751
DeskClock	(builtin)	2,311
1MobileMarket	1,000,000+	6,717
Convertor Pro ($$)	10,000+	806
No-frills CPU Control	1,000,000+	1,100

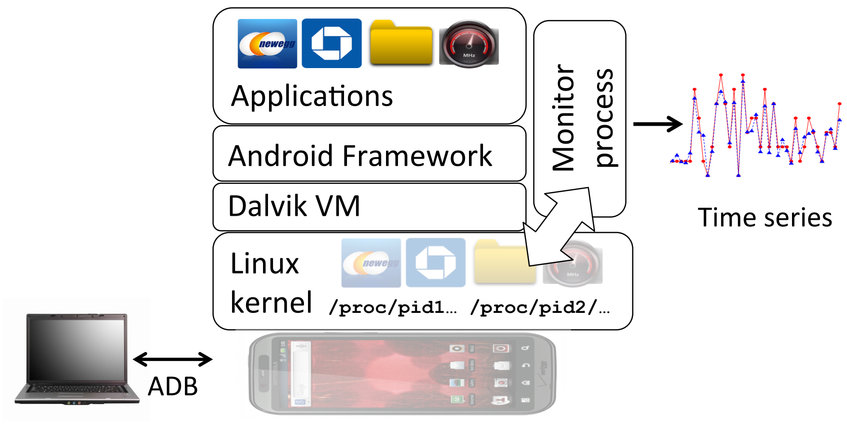

Fig. 16. Overview of our data collection process.

the monitor process samples the third entry in `/proc/pid/statm` of an application under test, and outputs a sequence of samples that will constitute the time series. We then analyze the time series using time series analysis, as will be explained shortly.

7.5. *Effectiveness*

We now quantify the effectiveness of our approach: we use time series complexity as a measure of effectiveness.

Time series complexity. Recall that attacks rely on predictability of app behavior as reflected in the `/proc/pid/statm` time series values: if the time series has high predictability (aka low complexity), the attack has a high chance of success. If the

Table 21. Evaluation results: activity transitions, time series entropy and transition time for each of the **S1**–**S4** strategies.

App	Activity Transitions	Permutation Entropy				Transition Time (msec)			
		S1	**S2**	**S3**	**S4**	**S1**	**S2**	**S3**	**S4**
Chase Mobile	Home → PrivacyOpt	0.332	0.361	**0.683**	0.729	544	248	**1,496**	864
	Home → ContactUs	0.454	0.524	**0.683**	0.349	688	360	**1,432**	1,312
	Home → FindBranch	0.332	0.235	**0.786**	0.332	664	248	**1,640**	784
Newegg Mobile	Main → ShoppingCart	0.256	0.406	**0.696**	0.129	1,088	904	**1,752**	1,088
	Main → WishListItem	0.361	0.332	**0.707**	0.372	1,520	88	**2,120**	1,520
	Main → Login	0.361	0.377	**0.682**	0.361	1,400	928	**2,008**	1,432
	Main → OrderHistory	0.457	0.358	**0.681**	0.364	1,464	112	**2,040**	1,208
	Main → MyPrsnlHomeCust	0.358	0.332	**0.722**	0.358	1,360	72	**1,920**	1,328
Browser	BrowserActivity → BrowserPrefsPg	0.332	0.332	**0.595**	0.657	136	24	**648**	672
GBC Emulator ($$)	MainActivity → EmulatorSettings	0.595	0.332	**0.595**	0.595	344	136	**1,040**	352
OI File Manager	FileManager → Preference	0.332	0.352	**0.595**	0.405	136	328	**656**	656
	FileManager → BookmarkList	0.372	0.332	**0.635**	0.657	40	248	**656**	672
Gallery3d	app. Gallery → sts . GallerySts .	0.522	0.391	**0.931**	0.489	136	88	**984**	160
VideoEditor	ProjectsActivity → VideoEditorActivity	0.333	0.403	**0.711**	0.333	128	144	**688**	688
Calendar	AllInOneActivity → CalendarSttgs	0.332	0.344	**0.620**	0.332	2,912	256	**3,536**	3,672
	AllInOneActivity → EditEvent	0.355	0.344	**0.724**	0.332	584	192	**2,080**	848
	AllInOneActivity → EventInfo	0.332	0.347	**0.637**	0.687	568	120	**776**	872
DeskClock	DeskClock → SettingsActivity	0.361	0.332	**0.661**	0.689	1,312	320	**1,960**	648
	DeskClock → worldclock . Cities	0.372	0.332	**0.726**	0.701	256	312	**752**	1,936
1Mobile Market	MainActivity → MyAppsInstdAct	0.129	0.333	**0.651**	0.801	1,280	1,296	**1,792**	2,064
	MainActivity → SettingsActivity	0.333	0.333	**0.633**	0.333	1,312	192	**560**	2,048
Convertor Pro ($$)	ProConvertAct → Settings	0.361	0.332	**0.595**	0.457	1,432	424	**2,104**	2,072
No-frills CPU Control	Main → Preferences	0.352	0.332	**0.763**	0.332	136	128	**688**	144
	Main → About	0.332	0.332	**0.332**	0.355	128	8	**664**	128
Average		0.361	0.351	**0.669**	0.465	815	299	**1,416**	1,132
% vs. **S1**			*-3%*	***+85%***	*+29%*		*-63%*	***+74%***	*+39%*

time series has low predictability (aka high complexity), the attacker will have a hard time inferring app behavior.

To measure time series complexity, we use the well-known *permutation entropy* (PE) metric[18] normalized so that $0 \leq PE \leq 1$. Here 0 represents no entropy, while 1 represents a random time series. Hence higher PE values are more desirable.

Time series results. Columns 3–7 in Table 21 show the results of the entropy measures. The main comparison is between strategy **S1**, that is the default Android implementation, and **S3** (our main approach). Note how the PE is consistently higher in **S3** than in **S1**. In the last row of the table, we show the average values across all activities. Note how PE increases from 0.361 on average (**S1**) to 0.669 (**S3**) – an 85% increase, which demonstrates that our proactive restart approach is effective at introducing randomness in the time series and consequently is effective at making the attacker's job harder. Strategies **S2** and **S4** are less effective if they are used in isolation (though **S4** has a 29% higher PE compared to **S1**). Nevertheless, note that if we were to combine **S2**, **S3**, and **S4** and randomly choose among them upon transitions, the compound strategy would fare even better than each of the individual strategies.

7.5.1. *Efficiency*

We now quantify the efficiency of our approach: are transitions taking longer with restart, and if so, how much longer?

The last four columns in Table 21 show the transition time, from the time the transition was initiated, to when it has completed (including restart time) for each of the four strategies. Compared to **S1**, **S3** increases transition time by 601 msec, i.e., a 74% increase. This exposes the cost-benefit trade-off: we pay the price of increasing transition time for the benefit of increasing attack resilience. Even though our implementation is not optimized, we believe that the added 601 msec are an acceptable overhead, especially when security is of high priority. Strategy **S2** takes less time than **S1** since no transition is involved. Strategy **S4** increases transition time by 317 msec, a 39% increase compared to **S1**.

8. Related Work

8.1. *Infrastructure and Reliability: Automated Exploration*

The work of Rastogi *et al.*[76] is most closely related to ours. Their system, named Playground, runs apps in the Android emulator on top of a modified Android software stack (TaintDroid); their end goal was dynamic taint tracking. They ran Playground on an impressive 3,968 apps and achieved 33% code coverage on average. The are several differences between their approach and A^{3}E. First, they run the apps on a modified software stack on top of the Android emulator, whereas we run apps on actual phones using an unmodified software stack. Second, Playground, just like our Depth-first Exploration, can miss activities — hence the need for Targeted Exploration which uses static analysis to find all the possible activities and entry points. Third, their GUI element exploration strategy is based on heuristics, ours is depth-first.

Memon *et al.*'s line of work on GUI testing for desktop applications[66,100,101] is centered around event-based modeling of the application to automate GUI exploration. Their approach models the GUI as an *event interaction graph* (EIG); the EIG captures the sequences of user actions that can be executed on the GUI. While the EIG approach is suitable for devising exploration strategies for GUI testing in applications with traditional GUI design, i.e., desktop applications, several factors pose complications when using it for touch-based smartphone apps, e.g., when using sensors.

Yang *et al.*[99] implemented a tool for automatic exploration called ORBIT. Their approach uses static analysis on the app's Java source code to detect actions associated with GUI states and then use a dynamic crawler (built on top of Robotium) to fire the actions. We focus on a different problem domain: large real-world apps for which the source code is not available.

Anand *et al.*[7] developed an approach named ACTEVE for concolic generation of events for testing Android apps whose source code is available. Their focus is on covering branches while avoiding the path explosion problem. ACTEVE generated test inputs for five small open source in 0.3–2.7 hours. Similarly, Jensen *et al.*[55] have used concolic execution to derive event sequences that can lead to a specific target state in an Android app. We believe that using concolic execution would allow us to increase coverage (especially method coverage), but it would require a symbolic execution engine robust enough to work on APKs of real-world substantial apps.

Monkey[10] is a testing utility provided by the Android SDK that can send a sequence of random and deterministic events to the app. Random events are effective for stress testing and fuzz testing, but not for systematic exploration; deterministic events have to be scripted, which involves effort, whereas in our case systematic exploration is automated. MonkeyRunner[9] is an API provided by the Android SDK which allows programmers to write Python test scripts for exercising Android apps. Similar to Monkey, scripts must be written to explore apps, rather than using automated exploration as we do.

Robotium[39] is a testing framework for Android that supports both black-box and white-box testing. Robotium facilitates interaction with GUI components such as menus, toasts, text boxes, etc., as it can discover these elements, fire related events, and generate test cases for exercising the elements. However, it does not permit automated exploration as we do.

Troyd[56] is a testing and capture-replay tool built on top of Robotium that can be used to extract GUI widgets, record GUI events and fire events from a script. We used parts of Troyd in our approach. However, Troyd cannot be used directly for either Targeted or Depth-first Exploration, as it needs input scripts for exercising GUI elements. Moreover, in its unmodified form, Troyd had a substantial performance overhead which slowed down exploration considerably — we had to modify it to reduce the performance overhead.

TEMA[84] is a collection of model-based testing tools which have been applied to Android. GUI elements form a state machine and basic GUI events are treated as keywords like events. Within this framework, test scripts can be designed and executed. In contrast, we extract a model either statically or dynamically and automatically construct test cases.

Android Ripper[6] is a GUI-based static and dynamic testing tool for Android. It uses a state-based approach to dynamically analyze GUI events and can be used to automate testing by separate test cases. Android Ripper preserves the state of the application where state is actually a tuple of a particular GUI widget and its properties. An input event triggers the change in the state and users can write test scripts based on the tasks that can modify the state. The approach works only on the Android emulator and thus cannot mimic sensor events properly like a real world application.

Several commercial tools provide functionality somewhat related to our approach, though their end-goals differ form ours. *Testdroid*[21] can record and run the tests on multiple devices. *Ranorex*[75] is a test automation framework. *Eggplant*[86] facilitates writing automated scripts for testing Android apps. *Framework for Automated Software Testing (FAST)*[91] can automate the testing process of Android apps over multiple devices.

8.2. *Infrastructure and Reliability: Record-and-Replay*

On the smartphone platform, the most powerful, and most directly related effort is our prior system Reran,[58] which has been used to record and replay GUI gestures in 86 out of the Top-100 most popular Android apps on Google Play. Reran does not require app instrumentation (hence it can handle gesture nondeterminism in apps that perform GUI rendering in native code, such as Angry Birds) or AF changes. Mosaic[43] extends Reran with support for device-independent replay of GUI events (note that our approach is device-independent as well). Mosaic has low overhead, typically less than 0.2%, and has replayed GUI events in 45 popular apps from Google Play. However, Reran and Mosaic have several limitations: they do not support critical functionality (network, camera, microphone, or GPS), required by many apps; they do not permit record-and-replay of API calls or event schedules; their record-and-replay infrastructure is manual, which makes it hard to modify or extend to other sensors.

Android test automation tools such as Android Guitar,[6,15] Robotium,[39] or Troyd[56] offer some support for automating GUI interaction, but require developers to extract a GUI model from the app and manually write test scripts to emulate user gestures. In addition to the manual effort required to write scripts, these tools do not support replay for sensors or schedules.

On non-smartphone platforms, record-and-replay tools have a wide range of applications: intrusion analysis,[33] bug reproducing,[67] debugging,[83] etc. Hardware-based[67,95] and virtual machine-based[33,80] replay tools are often regarded as whole-system replay. Recording at this low level, e.g., memory access order, thread scheduling, allows them to eliminate all non-determinism. However, these approaches require special hardware support or virtual machine instrumentation which might be prohibitive on current commodity smartphones.

Library-based approaches[30,53,77,96] record the non-determinism interaction between the program libraries and underlying operating system with a fixed interface. R2[40] extends them by allowing developers to choose which kinds of interfaces they want to replay by a simple annotation specification language. VALERA borrows this idea from R2 (which targets the Windows kernel API) but applies it to sensor-rich event-based Android.

8.3. *Security: Is the Ecosystem Moving in the "Right" Direction?*

Android permission characterization and effectiveness. Barrera *et al.*[29] introduced a self-organizing method to visualize permissions usage in different app categories. A comprehensive usability study of Android permissions was conducted through surveys in order to investigate Android permissions' effectiveness at warning users, which showed that current Android permission warnings do not help most users make correct security decisions.[12] Chia *et al.*[68] focused on the effectiveness of user-consent permission systems in Facebook, Chrome, and Android apps; they have shown that app ratings were not a reliable indicator of privacy risks.

Permission-related Android security. Enck *et al.*[88] presented a framework that read the declared permissions of an application at install time and compared it against a set of security rules to detect potentially malicious applications. Ongtang *et al.*[64] described a fine-grained Android permission model for protecting applications by expressing permission statements in more detail. Felt et al.[14] examined the mapping between Android API's and permissions and proposed Stowaway, a static analysis tool to detect over-privilege in Android apps. Permission re-delegation attacks were shown to perform privileged tasks with the help of an app with permissions.[13] Grace *et al.*[63] used Woodpecker to examined how the Android permission-based security model is enforced in pre-installed apps and stock smartphones. Capability leaks were found that could be exploited by malicious activities. DroidRanger was proposed to detect malicious apps in official and alternative markets.[98] Zhou *et al.* characterized a large set of Android malwares, e.g., accumulating fees on the devices by subscribing to premium services by abusing SMS_related Android permissions.[97] An effective framework was developed to defend against privilege-escalation attacks on devices.[78]

8.4. *Security: URL Risk*

Most security studies focus on malicious apps or OS-level exploits of good apps. There are several studies that focus on identifying malicious apps,[62,97,98] analyzing the source code and the OS behavior and permissions.[12,14,34,74,79] Then, there is a group of studies that study network traffic patterns.[42,72] Other efforts focus on user information leakage and attempt to detect the specific information that is being leaked.[61,69,87] Systems that rely on instrumenting the whole software stack, e.g., TaintDroid, can warn users when an Android app leaks sensitive data over the network.[90] web-oriented efforts, not necessarily focusing on smartphones, that evaluate and label websites[3,4] either focusing on malware, or considering a wider range of goodness based on user-feedback; we leverage such efforts to classify the websites here. However, our focus is different, in that we want to find out: (1) Which entities is the personal data potentially sent to (e.g., content providers, advertisers)? (2) Are these entities trusted? These questions are not answered in prior work.

8.5. *Security: App Profiling*

Falaki *et al.*[42] analyzed network logs from 43 smartphones and found commonly used app ports, properties of TCP transfer and the impact factors of smartphone performance. Furthermore, they also analyzed the diversity of smartphone usage, e.g., how the user uses the smartphone and apps.[46] Maier *et al.*[36] analyzed protocol usage, the size of HTTP content and the types of hand-held traffic. These efforts aid network operators, but they do not analyze the Android apps themselves. Recent work by Xu *et al.*[72] did a large scale network traffic measurement study on usage behaviors of smartphone apps, e.g., locality, diurnal behaviors and mobility patterns. Qian *et al.*[35] developed a tool named ARO to locate the performance and energy bottlenecks of smartphones by considering the cross-layer information ranging from radio resource control to application layer. Huang *et al.*[52] performed the measurement study using smartphones on 3G networks, and presented the app and device usage of smartphones. Falaki et al.[45] developed a monitoring tool SystemSens to capture the usage context, e.g., CPU and memory, of smartphone. Livelab[24] is a measurement tool implemented on iPhones to measure iPhone usage and different aspects of wireless network performance. Powertutor[59] focused on power modeling and measured the energy usage of smartphone. All these efforts focus on studying other layers, or device resource usage, which is different from our focus.

8.6. *Security: Moving Target Defense*

Application Restarts. Application restarts have been used in the past to remedy transient faults. Perkins *et al.*[71] used a reactive approach, named ClearView, that monitors an application's execution to learn application invariants, detect bugs or attacks, and upon detection automatically construct and apply a patch to heal the application. ClearView has been applied to Firefox. Ten exploits were presented to ClearView; upon repeated presentation, ClearView learned to identify each exploit and construct a patch against it. Our work is distantly related: our approach is proactive and attack-agnostic, as we do not perform monitoring, detection or patching, whereas ClearView uses sophisticated attack and bug-specific reactive techniques for invariant detection and patch construction.

Candea *et al.*[25] have proposed "microreboots" (rebooting small components instead of entire applications) as a recovery technique for Internet services. Our own prior work[16] has used online patch construction and application restart to provide self-healing capabilities — apps recovering from certain classes of transient and permanent faults — in Android apps. However, that approach was reactive, rather than proactive, and its goal was fault recovery rather than changing the attack surface. We are not aware of any work that uses restart as a cyber maneuver.

Android Side Channels Attacks and Defenses. Much work has been done on studying side channels. *Proc file systems* have been used for side-channel attacks. Zhang et al.[102] found that the ESP/EIP value can be used to infer keystrokes. Qian et al.[73] have used "sequence-number-dependent" packet counter side channels to infer TCP sequence number. In Memento,[54] the memory footprints were found to correlate with the web page the user is visiting. Zhou et al.[105] found 3 Android/Linux public resources to leak private information about location, disease, etc.. Chen et al.[26] proposed Activity inference attacks that can be applicable to all Android apps.

There are few effective defenses against the types of side-channel attacks. Lately, Zhang et al. have proposed to pause all suspicious background processes to stop them from gathering any data about the foreground app.[103] Such defense could be effective; however, it comes with a functionality cost — many background apps will not be able to function as designed. Another defense against the GUI state-manipulation attacks proposed by Bianchi et al. tries to provide explicit and secure indicators to keep the user informed about which app runs in the foreground at all times.[20] Such defense is tailored to attacks similar to Activity inference. In contrast, we believe application restart can be used as a general cyber maneuver against many types of side-channel attacks.

9. Conclusions and Future Work

We have presented a suite of infrastructural tools and approaches for improving Android's security and reliability. Our tools use program analysis, bytecode rewriting, multi-level monitoring and profiling to paint a clearer picture of Android app behavior; this can help Android developers and users better understand the security implications of running apps. In the last part of our paper we have shown how a simple restart-based approach can be used as an effective Moving Target Defense tool that provides higher resistance to known and unknown attacks.

We plan to extend this work in several directions.

Section 2: we plan to make $\mathsf{A^3E}$, the automated explorer, more effective by increasing the coverage it achieves, and also more efficient, to reduce the time it takes to explore an app.

Section 3: while VALERA captures and replays event order, in the current version VALERA does not capture thread nondeterminism or memory accesses. We plan to add support for capturing and replaying these, to enable reproducing data races. In addition, VALERA modifies the AF; enabling record-and-replay on stock Android without requiring AF changes would make the approach more portable and easier to use.

Section 4: one of the most serious issues we have found was over-privilege in pre-installed apps. A "sanitizer" that took a pre-installed app and would rewrite its bytecode to reduce privilege to the absolute minimum required (i.e., apply the principle of least privilege) would be a huge security advance for manufacturers and end-users alike.

Section 5: AURA discovers URLs either statically or dynamically. While static discovery has been shown more effective that dynamic discovery for our app dataset, its effectiveness might be a problem when apps use obfuscation, URL shortener or redirects, or dynamic URL generation/loading schemes. Dynamic discovery has scalability issues. Hence finding URLs effectively and efficiently remains a challenge that we plan to address.

Section 6: our study was based on 27 apps. An immediate extension would be to expand the range of apps, e.g., top-10 apps in each of the 30+ categories on Google Play. Next, a comparison between categories would reveal which apps are energy-intensive, privacy-sensitive, and so on.

Section 7: our MTD approach has been validated on a single attack (Activity Inference). Further experiments on how our approach withstands other kinds of attacks would increase confidence in our technique. Reducing transition time, while a technical – rather than research – challenge would go a long way toward making our approach more user-friendly.

Acknowledgments

This research was supported in part by NSF grant CNS-1630037. Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

1. App could allow troops to call in airstrikes, October 2013, http://www.militarytimes.com/article/20131016/NEWS04/310160005/App-could-allow-troops-call-airstrikes.
2. Google Play, https://play.google.com/store, September 2013.
3. VirusTotal, https://www.virustotal.com/en/#url, September 2013.
4. Web of Trust, http://www.mywot.com/, September 2013.
5. Army lists top 12 items in fiscal year 2016 budget request, April 2015, http://www.army.mil/article/145946/Army_lists_top_12_items_in_fiscal_year_2016_budget_request/.
6. D. Amalfitano, A. Fasolino, S. Carmine, A. Memon and P. Tramontana, Using GUI ripping for automated testing of android applications, in *ASE'12*.
7. S. Anand, M. Naik, M. J. Harrold and H. Yang, Automated concolic testing of smartphone apps, in *FSE '12*, pp. 1–11.
8. Android Developers, Android Emulator Limitations, http://developer.android.com/tools/devices/emulator.html#limitations.
9. Android Developers, MonkeyRunner, http://developer.android.com/guide/developing/tools/monkeyrunner_concepts.html.
10. Android Developers, UI/Application Exerciser Monkey, http://developer.android.com/tools/help/monkey.html.
11. Android Police, Massive Security Vulnerability in HTC Android Devices, http://www.androidpolice.com/2011/10/01/massive-security-vulnerability-in-htc-android-devices, October 2011.

12. A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin and D. Wagner, Android Permissions: User Attention, Comprehension, and Behavior, in *SOUPS*, 2012.
13. A. P. Felt, H. Wang, A. Moshchuk, S. Hanna and E. Chin, Permission Re-Delegation: Attacks and Defenses, in *USENIX Security Symposium*, 2011.
14. A. P. Felt, E. Chin, S. Hanna, D. Song and D. Wagner, Android Permissions Demystified, in *ACM CCS*, 2011.
15. Atif Memon, GUITAR, August 2012, guitar.sourceforge.net/.
16. Md. T. Azim, I. Neamtiu and L. M. Marvel, Towards self-healing smartphone software via automated patching, in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ACM, 2014, pp. 623–628.
17. B. Krishnamurthy and C. E. Willis, Privacy diffusion on the web: A longitudinal perspective, in *WWW*, 2009.
18. C. Bandt and B. Pompe, Permutation entropy: a natural complexity measure for time series, *Physical Review Letters* **88**(17) (2002) 174102.
19. P. Bhattacharya, L. Ulanova, I. Neamtiu and S. C. Koduru, An empirical analysis of the bug-fixing process in open source android apps, in *CSMR'13*.
20. A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel and G. Vigna, What the App is That? Deception and Countermeasures in the Android User Interface, in *IEEE Symposium on Security and Privacy*, 2015.
21. Bitbar, Automated Testing Tool for Android — Testdroid, January 2013, http://testdroid.com/.
22. M. Böhmer, B. Hecht, J. Schöning, A. Krüger and G. Bauer, Falling asleep with angry birds, facebook and kindle: a large scale study on mobile application usage, in *MobileHCI '11*, pp. 47–56.
23. C. Labovitz, S. Iekel-Johnson, D. McPherson, J. Oberheide and F. Jahanian, Internet inter-domain traffic, in *ACM SIGCOMM*, 2010.
24. C. Shepard, A. Rahmati, C. Tossell, L. Zhong and P. Kortum, LiveLab: Measuring Wireless Networks and Smartphone Users in the Field, in *HotMetrics*, 2010.
25. G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman and A. Fox, Microreboot: A technique for cheap recovery, pp. 31–44.
26. Qi Alfred Chen, Zhiyun Qian, and Zhuoqing Morley Mao. Peeking into your app without actually seeing it: UI state inference and novel android attacks, in *Proceedings of the 23rd USENIX Security Symposium*, San Diego, CA, USA, August 20–22, 2014, pp. 1037–1052.
27. CNET, Android reclaims 61 percent of all U.S. smartphone sales, May 2012, http://news.cnet.com/8301-1023_3-57429192-93/android-reclaims-61-percent-of-all-u.s-smartphone-sales/.
28. B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen and R. Koschke, A systematic survey of program comprehension through dynamic analysis, *IEEE Transactions on Software Engineering*, pp. 684–702, 2009.
29. D. Barrera, H. G. Kayacik, P. C. van Oorschot and A. Somayaji, A Methodology for Empirical Analysis of Permission-based Security Models and its Application to Android, in *ACM CCS*, 2010.
30. D. Geels, G. Altekar, S. Shenker and I. Stoica, Jockey: a user-space library for record-replay debugging, in *USENIX ATC'06*.

31. D. C. Hoaglin, F. Mosteller and J. W. Tukey, *Understanding Robust and Exploratory Data Analysis* (Wiley, 1983).
32. B. Dolan, FDA approves Mobisante's smartphone ultrasound, http://www.mobihealthnews.com/10165/fda-approves-mobisantes-smartphone-ultrasound.
33. G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai and P. M. Chen, Revirt: enabling intrusion analysis through virtual-machine logging and replay, in *OSDI'02*.
34. E. Chin, A. P. Felt, K. Greenwood and D. Wagner, Analyzing Inter-Application Communication in Android, in *ACM MobiSys*, 2011.
35. F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen and O. Spatscheck, Profiling Resource Usage for Mobile apps: a Cross-layer Approach, in *ACM MobiSys*, 2011.
36. G. Maier, F. Schneider and A. Feldmann, A First Look at Mobile Hand-held Device Traffic, in *PAM*, 2010.
37. Gartner, Inc., Gartner Highlights Key Predictions for IT Organizations and Users in 2010 and Beyond, January 2010, http://www.gartner.com/it/page.jsp?id=1278413.
38. Gartner, Inc., Gartner Says Worldwide PC Shipment Growth Was Flat in Second Quarter of 2012, July 2012, http://www.gartner.com/it/page.jsp?id=2079015.
39. Google Code, Robotium, August 2012, http://code.google.com/p/robotium/.
40. Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek and Z. Zhang, R2: An application-level kernel for record and replay, in *OSDI'08*.
41. J. Guyn, Facebook users give iPhone app thumbs down, *Los Angeles Times*, July 21 2011, http://latimesblogs.latimes.com/technology/2011/07/facebook-users-give-iphone-app-thumbs-down.html.
42. H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin, A First Look at Traffic on Smartphones, in *ACM IMC*, 2010.
43. M. Halpern, Y. Zhu and V. J. Reddi, Mosaic: Cross-platform user-interaction record and replay for the fragmented android ecosystem, in *ISPASS'15*.
44. S. Hao, D. Li, W. G. J. Halfond and R. Govindan, Estimating android applications' CPU energy usage via bytecode profiling, in *2012 First International Workshop on Green and Sustainable Software* (*GREENS*), pp. 1–7, 2012.
45. H. Falaki, R. Mahajan and D. Estrin, SystemSens: A Tool for Monitoring Usage in Smartphone Research Deployments, in *ACM MobiArch*, 2011.
46. H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan and D. Estrin, Diversity in Smartphone Usage, in *ACM MobiSys*, 2010.
47. C. Hu and I. Neamtiu, Automating GUI testing for android applications, in *AST '11*, pp. 77–83, 2011.
48. Y. Hu, T. Azim and I. Neamtiu, Versatile yet lightweight record-and-replay for android, in *OOPSLA'15*.
49. Y. Hu and I. Neamtiu, Fuzzy and cross-app replay for smartphone apps, in *AST'16*.
50. Y. Hu, I. Neamtiu and A. Alavi, Automatically verifying and reproducing event-based races in android apps, in *Proceedings of the 25th International Symposium on Software Testing and Analysis* (*ISSTA 2016*), Saarbrücken, Germany, July 18–20, 2016, pp. 377–388, 2016.
51. IDC, Android and iOS Surge to New Smartphone OS Record in Second Quarter, According to IDC, August 2012, http://www.idc.com/getdoc.jsp?containerId=prUS23638712.

52. J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang and P. Bahl, Anatomizing app Performance Differences on Smartphones, in *ACM MobiSys*, 2010.
53. J. Steven, P. Chandra, B. Fleck, and A. Podgurski, jRapture: A capture/replay tool for observation-based testing, in *ISSTA'00*.
54. S. Jana and V. Shmatikov, Memento: Learning Secrets from Process Footprints, in *IEEE Symposium on Security and Privacy*, pp. 143–157, 2012.
55. C. S. Jensen, M. R. Prasad and A. Møller, Automated testing with targeted event sequence generation, in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pp. 67–77, 2013.
56. J. Jeon and J. S. Foster, Troyd, January 2013, https://github.com/plum-umd/troyd.
57. J. Jeon, K. Micinski and J. S. Foster, Redexer, http://www.cs.umd.edu/projects/PL/redexer/index.html.
58. L. Gomez, I. Neamtiu, T. Azim and T. Millstein, Reran: Timing- and touch-sensitive record and replay for android, in *ICSE '13*.
59. L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. Dick, Z. M. Mao and L. Yang, Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones, in *CODES+ISSS*, 2010.
60. M. Egele, C. Kruegel, E. Kirda and G. Vigna, Detecting Privacy Leaks in iOS apps, in *NDSS*, 2011.
61. M. Egele, C. Kruegel, E. Kirda, and G. Vigna, PiOS: Detecting Privacy Leaks in iOS Applications, in *NDSS*, 2011.
62. M. Grace, Y. Zhou, Q. Zhang, S. Zou, X. Jiang, RiskRanker: Scalable and Accurate Zero-day Android Malware Detection, in *ACM MobiSys*, 2012.
63. M. Grace, Y. Zhou, Z. Wang, and X. Jiang, Systematic Detection of Capability Leaks in Stock Android Smartphones, in *NDSS*, 2012.
64. M. Ongtang, S. McLaughlin, W. Enck and P. McDaniel, Semantically Rich Application-Centric Security in Android, in *ACSAC*, 2009.
65. P. Maiya, A. Kanade and R. Majumdar, Race detection for android applications, in *PLDI'14*.
66. A. M. Memon, An event-flow model of GUI-based applications for testing, *Software Testing, Verification and Reliability*, pp. 137–157, 2007.
67. S. Narayanasamy, G. Pokam and B. Calder, Bugnet: Continuously recording program execution for deterministic replay debugging, in *ISCA '05*.
68. P. H. Chia, Y. Yamamoto and N. Asokan, Is this App Safe? A Large Scale Study on Application Permissions and Risk Signals, in *WWW*, 2012.
69. 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 *ACM CCS*, 2011.
70. P. Pearce, A.P. Felt, G. Nunez and D. Wagner, AdDroid: Privilege Separation for Applications and Advertisers in Android, in *ACM AsiaCCS*, 2012.
71. J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst and M. Rinard, Automatically patching errors in deployed software, in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (*SOSP '09*), pp. 87–102, New York, NY, USA, ACM 2009.

72. Q. Xu, J. Erman, A. Gerber, Z. M. Mao, J. Pang, and S. Venkataraman, Identify Diverse Usage Behaviors of Smartphone Apps, in *IMC*, 2011.
73. Z. Qian, Z. M. Mao and Y. Xie, Collaborative tcp sequence number inference attack: how to crack sequence number under a second, in *CCS*, pp. 593–604, 2012.
74. R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia and X. Wang, Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones, in *NDSS*, 2011.
75. Ranonex, Android Test Automation - Automate your App Testing, January 2013, http://www.ranorex.com/mobile-automation-testing/android-test-automation.html.
76. V. Rastogi, Y. Chen and W. Enck, Appsplayground: automatic security analysis of smartphone applications, in *CODASPY*, pp. 209–220, 2013.
77. Michiel, Ronsse and Koen De Bosschere, RecPlay: A Fully Integrated Practical Record/Replay System, *ACM Trans. Comput. Syst.* **17**(2) (1999) 133–152.
78. S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A. Sadeghi and B. Shastry, Towards Taming Privilege-Escalation Attacks on Android, in *NDSS*, 2012.
79. S. Fahl, M. Harbach, T. Muders, L. Baumgartner, B. Freisleben and M. Smith, Why eve and mallory love android: an analysis of android SSL (in)security, in *ACM CCS*, 2012.
80. S. T. King, G. W. Dunlap and P. M. Chen, Debugging operating systems with time-traveling virtual machines, in *USENIX ATC'05*.
81. Z. Shan, I. Neamtiu, Z. Qian and D. Torrieri, Proactive restart as cyber maneuver for android, in *Military Communications Conference* (*MILCOM 2015*), IEEE 2015, pp. 19–24, 2015.
82. SourceForge, Android GUITAR, August 2012, http://sourceforge.net/apps/mediawiki/guitar/index.php?title=Android_GUITAR.
83. S. M. Srinivasan, S. Kandula, C. R. Andrews and Y. Zhou, Flashback: a lightweight extension for rollback and deterministic replay for software debugging, in *USENIX ATC'04*.
84. T. Takala, M. Katara, and J. Harty, Experiences of system-level model-based GUI testing of an Android application, in *ICST '11*, pp. 377–386.
85. T. Azim and I. Neamtiu, Targeted and depth-first exploration for systematic testing of android apps, in *OOPSLA'13*.
86. TestPlant, eggPlant for mobile testing, January 2013, http://www.testplant.com/products/eggplant/mobile/.
87. W. Enck, D. Octeau, P. McDaniel and S. Chaudhuri, A Study of Android Application Security, in *USENIX Security Symposium*, 2011.
88. W. Enck, M. Ongtang and P. McDaniel, On Lightweight Mobile Phone Application Certification, in *ACM CCS*, 2009.
89. W. Enck, P. Gilbert, 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, in *OSDI*, pp. 393–407, 2010.
90. W. Enck, P. Gilbert, 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, in *OSDI*, 2010.
91. W. River, Wind River Framework for Automated Software Testing, January 2013, http://www.windriver.com/announces/fast/.

92. X. Wei, L. Gomez, I. Neamtiu and M. Faloutsos, Permission evolution in the android ecosystem, in *Proceedings of the 28th Annual Computer Security Applications Conference* (*ACSAC '12*), pp. 31–40, New York, NY, USA, ACM 2012.
93. X. Wei, I. Neamtiu and M. Faloutsos, Whom does your android app talk to? in *2015 IEEE Global Communications Conference* (*GLOBECOM 2015*), San Diego, CA, USA, December 6–10, 2015, pp. 1–6.
94. X. Wei, L. Gomez, I. Neamtiu and M. Faloutsos, ProfileDroid: Multi-layer Profiling of Android Applications, in *ACM MobiCom*, 2012.
95. M. Xu, R. Bodik and M. D. Hill, A "flight data recorder" for enabling full-system multiprocessor deterministic replay, in *ISCA '03*, pp. 122–135.
96. Y. Saito, Jockey: a user-space library for record-replay debugging, in *AADEBUG'05*.
97. Y. Zhou and X. Jiang, Dissecting Android Malware: Characterization and Evolution, in *IEEE S &P*, 2012.
98. Y. Zhou, Z. Wang, W. Zhou and X. Jiang, Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets, in *NDSS*, 2012.
99. W. Yang, M. Prasad and T. Xie, A grey-box approach for automated GUI-model generation of mobile applications, in *FASE'13*, pp. 250–265.
100. X. Yuan and A. M. Memon, Using GUI run-time state as feedback to generate test cases, in *ICSE '07*, pp. 396–405, 2007.
101. X. Yuan and A. M. Memon, Generating event sequence-based test cases using GUI run-time state feedback, *IEEE Transactions on Software Engineering*, pp. 81–95, 2010.
102. K. Zhang and X. Wang, Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems, in *USENIX Security Symposium*, pp. 17–32, 2009.
103. N. Zhang, K. Yuan, M. Naveed, X. Zhou and X. Wang, Leave Me Alone: App-level Protection Against Runtime Information Gathering on Android, in *IEEE Symposium on Security and Privacy*, 2015.
104. W. Zhou, Y. Zhou, X. Jiang and P. Ning, Detecting repackaged smartphone applications in third-party android marketplaces, in *CODASPY*, pp. 317–326, 2012.
105. X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter and K. Nahrstedt, Identity, Location, Disease and More: Inferring Your Secrets from Android Public Resources, in *CCS*, pp. 1017–1028, 2013.