ZapDroid: Managing Infrequently Used Applications on Smartphones

Indrajeet Singh
UC Riverside
singhi@cs.ucr.edu

Srikanth V. Krishnamurthy
UC Riverside
krish@cs.ucr.edu

Harsha V. Madhyastha
University of Michigan
harshavm@umich.edu

Iulian Neamtiu
UC Riverside
neamtiu@cs.ucr.edu

ABSTRACT

User surveys have shown that a typical user has over a hundred apps on her smartphone [1], but stops using many of them. We conduct a user study to identify such unused apps, which we call zombies, and show via experiments that zombie apps consume significant resources on a user's smartphone and access her private information. We then design and build *ZapDroid*, which enables users to detect and silo zombie apps in an effective way to prevent their undesired activities. If and when the user wishes to resume using such an app, *ZapDroid* restores the app quickly and effectively. Our evaluations show that: (i) *ZapDroid* saves twice the energy from unwanted zombie app behaviors as compared to apps from the Play Store that kill background unwanted processes, and (ii) it effectively prevents zombie apps from using undesired permissions. In addition, *ZapDroid* is energy-efficient, consuming $< 4\%$ of the battery per day.

ACM Classification Keywords
D.4.9 Operating Systems: Systems Programs and Utilities

Author Keywords
Smartphones; energy; privacy

INTRODUCTION

The Google Play Store has more than 1.3 million apps [22], and the number of app downloads is roughly 1 billion per month [15]. However, after users interact with many such apps for an initial period following the download, they almost never do so again. Statistics indicate that for a typical app, less than half of the people who downloaded it use it more than once [13]. Reports also suggest that more than 86 % of users do not even revisit an app, a day after the initial download [30]. Uninstall rates of apps however (longer term), of about 15 to 18 % are considered high [26]. This means that users often leave installed apps on their phones.

UbiComp '15, September 7-11, 2015, Osaka, Japan.
Copyright 2015 © ACM 978-1-4503-3574-4/15/09...$15.00.
http://dx.doi.org/10.1145/2750858.2807550

More generally, users may only interact with some downloaded apps infrequently (i.e., not use them for prolonged periods). These apps continue to operate in the background and have significant negative effects (e.g., leak private information or significantly tax resources such as the battery). Unfortunately, users are often unaware of such app activities. We call such seldom-used apps, which indulge in undesired activities, "zombie apps."

In this paper, we seek to build a framework, *ZapDroid*, to identify and subsequently quarantine such zombie apps to stop their undesired activities. Since a user can change her mind about whether or not to use an app, a zombie app must be restored quickly if the user chooses.[1]

The classification of an app as a zombie app is inherently subjective. An app unused for a prolonged period should be classified as a zombie app if its resource usage during the period is considered significant and/or if its access of private data is deemed serious. Thus, instead of automatically classifying zombie apps, we seek to empower the user by exporting the information that she would need to make this decision. Moreover, the way in which a zombie app should be quarantined depends on whether the user is likely to want to use the app again in the future (e.g., a gaming app that the user tried once and decided is not interesting vs. a VoIP app that the user uses infrequently). The apps that a user is likely to use again fairly soon must not be fully uninstalled; real time restoration (when needed) may be difficult if there is no good network connectivity. We seek to enable users to deal with these different scenarios appropriately.

Challenges: We address many challenges en route designing and building *ZapDroid*. First, to motivate the need for *ZapDroid*, we ask the question: "How often do users download apps and leave them on their phones, and how do these apps adversely affect the user in terms of consuming phone resources and privacy leakage?" We address this challenge via an extensive user study. Next, we ask "How can we detect background apps that either consume high resources or violate privacy in a *lightweight* manner ?" Such apps are the candidates for being zombie

[1]For example, [12] reports that there are over 300 million installs of Skype, but the number of active users daily is only 4.9 million. Inactive users of Skype may want to disengage the app to prevent it from using network/energy resources. However, they may want quick restoration when needed.

apps. Continuous app monitoring (e.g., [49]) can be too resource-intensive to be practical. Further, application-level implementations are infeasible since Android does not allow any app to track the permission access patterns of other apps. The third challenge is to effectively quarantine apps, i.e., "How can we design effective methods to ensure that zombie apps are quarantined and remain in that state unless a user wants them restored?" With current approaches, apps' background activities are constrained only temporarily [32], until they are woken up due to time-outs or external stimuli [6, 14]. Finally, " How can we restore previously quarantined apps in a timely way, even under conditions of poor network connectivity (if the user desires)?" The restored app must be in the same state that it was in prior to the quarantine. Reinstalls from the Google Play Store can be hard if network connectivity is poor and hence, should not be invoked when it is highly likely that the user will restore the app. Further, clean uninstalls can result in loss of application state.

Contributions: Our framework, *ZapDroid*, addresses the above challenges and allows users to effectively manage infrequently used apps. In designing and building *ZapDroid*, we make the following contributions.

- ***Showcase the unwanted behaviors of candidate zombie apps:*** We conduct a month-long study where we enlist 80 users on Amazon's Mechanical Turk to download an app (`TimeUrApps`) we develop. `TimeUrApps` identifies (other) apps that have not been used for the month, on the users' phones. Once we identify these apps, we undertake an in-house, comprehensive experimental study to understand their behaviors when they are not being actively used. We find that a zombie app on a typical user's phone (the median user in our targeted experiments) could consume as much as 58 MB of bandwidth and more than 20% of the total battery capacity in a day. Further, many of such apps access information such as the user's location and transmit this over the network.
- ***Identify candidate zombie apps that are most detrimental to the user's device:*** We design mechanisms that are integrated within the Android OS (we make changes to the underlying Android Framework's activity management, message passing, and resource management components) to track (i) a user's interactions with the apps on her device to identify unused apps, and (ii) the resources consumed and the private information accessed by these apps to determine candidate zombie apps, from which the user can choose to quarantine those she considers to be zombie apps.
- ***Dynamically revoke permissions from zombie apps, or offload them to external storage:*** The quarantine module of *ZapDroid* is invoked based on user input. She has to categorize a zombie app as either "likely to restore" or "unlikely to restore"; the two categories are quarantined differently. For the first category, only permissions enjoyed by the zombie app are revoked but all relevant data/binaries are stored on the device itself. For the second category, the associated data/binaries are removed from the device and user-specific app state is moved to either the cloud or to a different device (a desktop) owned by the user; the transfers occur when there is good network connectivity (e.g., WiFi coverage or a USB cable).
- ***Restore an app with all its permissions if the user desires:*** *ZapDroid* restores a zombie app on the user's device if she so desires. The state of the app is identical to that prior to the quarantine. For the "likely to restore" category of apps, the restoration time is $< 6ms$. For the "unlikely to restore" category, restoration depends on the network connectivity to where the app was stored during the quarantine and is typically on the order of a few seconds.

We evaluate *ZapDroid* via extensive measurements on 5 different Android smartphones (from 4 vendors). We show that the overhead of *ZapDroid* is low ($< 4\%$ of the battery is consumed per day). We show that *ZapDroid* saves more than $2\times$ the energy expended due to zombie app activities, as compared to other popular apps on the Google Play Store used to kill undesired background processes; further, unlike these apps, it prevents access to undesired permissions by the zombie apps.

Note that *ZapDroid* does not require changes to an external cloud store (for quarantine or restoration); all modifications are made only in the Android OS. We envision that the features of *ZapDroid* will be useful in general, and our hope is that this could lead to an integration of the functions within the Android OS.

UNDERSTANDING ZOMBIE APPS

We begin with an in-depth measurement study that has two main goals: (a) revealing zombie apps, via an IRB-approved user study, and (b) profiling zombie apps to quantify their adverse effects both in terms of resource consumption as well as privacy leaks.

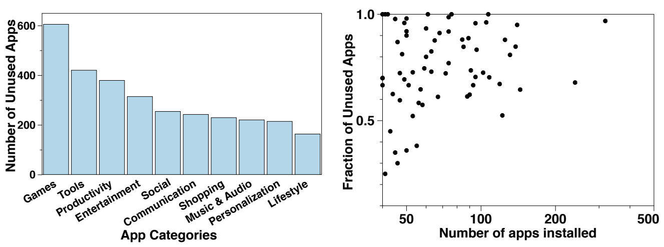
User Studies

We undertake a large-scale user study to identify apps that are installed but not used. We build an app, `TimeUrApps`,and post it to the Google Play Store. We solicit volunteers on Amazon Mechanical Turk to install `TimeUrApps`. `TimeUrApps` records the following for 30 days: **(i)** the apps installed on users' devices, and **(ii)** the timestamps of all events where an app is switched to the foreground from the background or vice versa. It implicitly starts the Accessibility Services [3] upon activation, which facilitates the collection of this information. This allows the detection of apps that have been inactive for extended periods.

To be eligible for our study, we required that a user must have at least 40 third-party apps installed. This number was motivated by a Nielsen study [20], which reports that the average number of third-party apps on a user's smartphone is 41 (although many users can have a much higher number of such apps [13]). A total of 87 users downloaded `TimeUrApps`. We filtered out 7 users that

Category	Types of Apps
Games	Role playing and turn-based games
Tools	Memory cleaners and task managers
Productivity	Barcode scanners and cloud drives
Entertainment	Media streaming and ticket sales
Social	Social networks and event planning

Table 1: Types of apps in the top 5 categories.



(a) Categories of apps (b) Fraction of unused apps

Figure 1: App statistics from the user study.

installed new apps just prior to downloading our app, to reach the required app count of 40 (so that they could claim the reward that we promised). This left us with 80 users. The data collected by `TimeUrApps` from each user's phone was transferred to our servers using WiFi as long as access was available within a five day period after collection. If no WiFi access was available for 5 days, the data was transferred on the cellular network. All the apps that were unused for a period of a month are considered potential zombie apps. We select the top 5 categories of unused apps (Table 1) based on this data set and perform a more in-depth study. The categories are based on the definitions in the Google Play Store.

The number of apps in various categories that we observed in our dataset is plotted in Figure 1a. We see that more than 1,000 unique apps were not used in the one-month test period. We also examine whether the number of unused apps on a device depends on the total number of apps installed. In Figure 1b, we show the fraction of apps that were never used, for each user, in the month. Interestingly, the results suggest that regardless of how many apps were installed on a phone, a user rarely interacts with more than half of them.

Offline Measurement Methodology

Once we identify unused apps from our users, we perform offline measurements to determine which of these consume significant resources or access/leak private information while running in the background. We measure the resources consumed in terms of the battery drain, the network bandwidth, and time spent on the CPU. Note that we did not measure these on the study subjects' phones since this needs active monitoring (rooting the phones) and transferring large volumes of data.

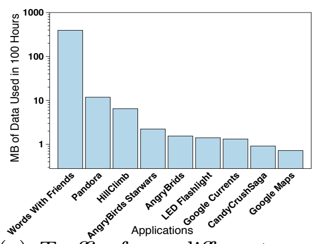
Scenarios considered: Our measurement studies were conducted on five different smartphones, from four different vendors: a Moto X (Motorola), a Nexus 4 and a Nexus 5 (LG), a Samsung Galaxy S4, and an HTC One. *First,* we did a quick examination to find the apps that consumed the highest resources. We then installed these apps in isolation, and let them run in background mode without user intervention for a period of 100 hours. Our goal is to characterize individual app behavior. *Second,* we chose 15 users at random from our 80 volunteers, each with a different number of unused apps (ranging from 7 to 54), and installed all the unused apps from a user profile on a phone. We then executed all these apps in background mode without user intervention, to quantify the collective impact of the apps. In this section, we only show subsets of our results due to space constraints.

For some of the apps, we took additional steps to ensure their proper execution. For example, the Facebook app will not execute unless the user has an account and has logged onto it on the device (many users who had Facebook installed never interacted with it during our month-long study!). A second example was the Angry Birds game; unless the users finished the first level, the app would not have an initial score to check against other users and would not communicate with a server. In this case, we assume that a user who installed Angry Birds has completed at least one level. Our rendering in the above cases provides a conservative estimate of resources consumed by the apps; a more complex (and possibly realistic) profile will incur higher resource drain.

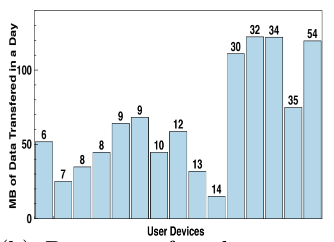
Measuring resource consumption: Our smartphones transfer content over WiFi (we have some limited experiments over LTE as discussed later). Each device connects to a Man-in-the-Middle (MitM) Proxy, which logs every IP packet that the device sends out. On the Android devices themselves, we (a) embed a certificate generated by our proxy as a "root certificate" to ensure that the proxy can decode all packets sent by the device, and (b) install `tcpdump` to monitor the traffic the device sends out on the local network, e.g., UDP broadcasts. We used `PowerTutor`[49] to estimate the CPU usage and energy consumption by each app. The readings in Joules are converted to a battery percentage. For example, the Samsung Galaxy S4 has a a rating of 9.88Wh (2600mAh@3.8V [24]), which corresponds to 9.88×3600 J. The percentage of battery power is expressed relative to this rating. Note that `PowerTutor` is not used with *ZapDroid*; it is only used in our measurement study.

Permission access patterns: Since Android does not automatically allow us to log the permissions accessed by an app, we root one of our phones and install *ZapDroid* (described later). This allowed us to log the default permissions that were accessed by these applications.

Measuring space consumed by zombie apps: As an auxiliary resource, we measure the disk space consumed by each zombie app on any user's smartphone. There are two types of data being stored: (i) *App Binary*: This is user-independent data, which includes the app binary downloaded from the Google Play Store (`.apk`) and any additional data that the app will need for it to function; the Google Play Store puts a limit of 50MB on the former and 2GB on the latter [17], (ii) *App Data*: This is user-dependent data, which is *not* content created by the user herself (e.g., photos) but the byproduct of her interaction with the app (e.g., temporary files); we only focus

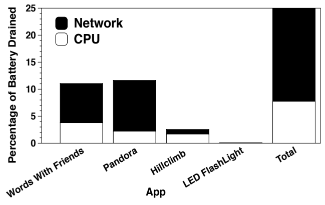


(a) Traffic from different zombie apps

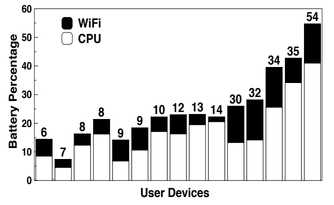


(b) Data transferred per user (no. of zombie apps on bars)

Figure 2: Network transfers by zombie apps.



(a) Energy consumed by zombie apps on a Nexus 5



(b) Energy consumed per user (no. of zombie apps on bars)

Figure 3: Energy consumption by zombie apps.

on the latter because the former cannot be classified as something that the user would want to get rid of. Since our user study did not provide us with any information on the *App data* (the phones were not rooted), we distributed our app to 25 volunteer students with rooted Android phones and gathered this data separately.

Measurement Results and Inferences

Next, we present our measurements to showcase the impact of zombie apps (any app that was unused for the month is considered a zombie app) on a user's device.

Bandwidth impact: In Figure 2a, we quantify the network traffic generated by some of the popular zombie apps in our dataset. Many of these consume bandwidth due to advertisements. Games like *Words with Friends*, in addition, actively try to find new games that are likely to interest the user. These apps therefore consumed an inordinate amount of network bandwidth, due to continuous synchronization with remote servers. Worse, even if the user ignored or disabled the app's notifications it continued to perform the activity in the background. The figure also shows that a zombie app could consume more than 1 MB of bandwidth over a 24-hour period; thus, over the same period, a user with 20 zombie apps could consume 20 MB of bandwidth; most of our 80 volunteer users had at least 20 apps that remained unused for the month. If the WiFi is turned off, these communications happen over the cellular network. We verified this over a 12-hour period. During this time, for example, approximately 38 MB was transferred over LTE due to Words with Friends (similar to WiFi). This hurts the user when mobile operators impose limits on cellular data usage.

Our next experiment is performed with the second scenario described earlier, and captures the collective behaviors of apps for a randomly selected set of users. The network transfers due to each user's zombie apps are shown in Figure 2b. We observe that the collective group of zombie apps on some users' phones could transfer more than 100 MB of data per day.

CPU consumption: For many zombie apps, CPU spikes caused by the apps correlate with network transfers. There are, however, some zombie apps that continuously consume CPU cycles in the background, even when not indulging in network activities. One such app is HillClimb, which, as revealed in our `strace`-based [27] inspection, updates its internal database that stores cached advertisements; the app also records the levels that have been completed by the user. Since this resource expense is implicitly captured in our energy results, we omit the details due to space constraints.

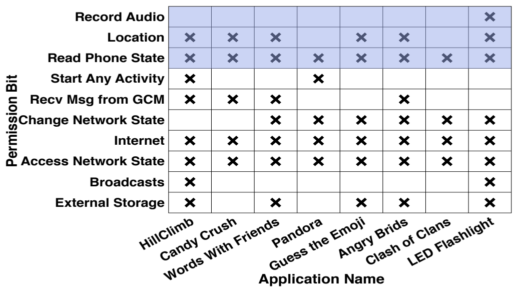
Energy drain: In Figure 3a, we show the battery drain due to both network and CPU activities by popular zombie apps (the same as in Figure 2a) on a Google Nexus 5, which has a battery capacity of 2300 mAh at 3.7V. We see that 4 of the zombie apps consumed about 6% of the battery capacity per day ($\approx$ 25% of the battery over the 100-hour period). Note that this measurement was conducted over WiFi; over the cellular network the energy consumed could have been three times as much [43]. Finally, the collective impact of a larger number of such zombie apps (as is the case on our volunteers' phones) will be even worse. In Figure 3b, we see that over the 24-hour period, the zombie apps installed on a typical user's phone consumed about 24% of a Galaxy S4's battery power on average (median of about 22%).

Use of undesirable permissions and privacy leaks: From Figure 4, we observe that almost all zombie apps access the phone's state which allows them to get a unique identifier associated with the device. This is mostly used for advertisement delivery and for tracking the end user. However, a zombie app can use this permission to track who a user may be calling [29]. In addition, most zombie apps requested access to the user's location, or a permission to modify the network connectivity (e.g., switch access points with WiFi, or disconnect from a network) of the device. Note that *all* the zombie apps shown accessed the Internet in the background. The part of the figure that is shaded, indicates permission bits that leak privacy; we verified that some of this information was actually communicated over the network (e.g., HillClimb accessed and transferred the unique device identifier – Read Phone State – over the network). These results indicate that there is a significant risk of privacy leakages when a user has zombie apps on her smartphone.

Storage: From Figure 5, we see that the amount of space taken up by an app is dominated by the size of the binary (and additional user-independent data required by the app) and not the data created by the user. Thus, if storage is a concern for users, stopping the execution of zombie apps is inadequate by itself.

AN OVERVIEW OF ZAPDROID

ZapDroid's goal is to eliminate the adverse effects of infrequently used apps on a user's device by effectively quarantining them. However, it also seeks to quickly restore such apps if the user later wishes to interact with



Permission Bit	HillClimb	Candy Crush	Words With Friends	Pandora	Guess the Emoji	Angry Brids	Clash of Clans	LED Flashlight
Record Audio								×
Location	×	×	×		×	×		×
Read Phone State	×	×	×	×	×	×	×	×
Start Any Activity	×			×				
Recv Msg from GCM	×	×	×			×		
Change Network State			×	×	×	×	×	×
Internet	×	×	×	×	×	×	×	×
Access Network State	×	×	×	×	×	×	×	×
Broadcasts	×							×
External Storage	×		×		×	×		×

Application Name

Figure 4: Permissions used by zombie apps.

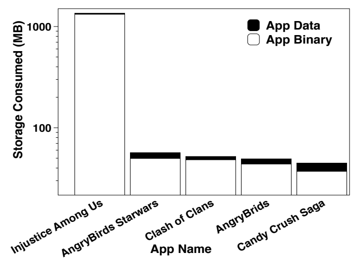


Figure 5: Storage consumed by zombie apps.

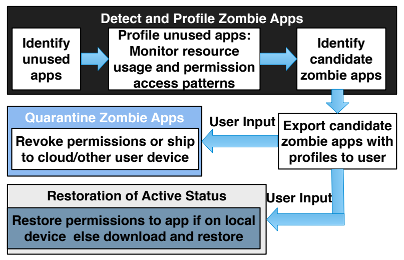

Detect and Profile Zombie Apps
Identify unused apps
Profile unused apps: Monitor resource usage and permission access patterns
Identify candidate zombie apps
Export candidate zombie apps with profiles to user
User Input
Quarantine Zombie Apps
Revoke permissions or ship to cloud/other user device
User Input
Restoration of Active Status
Restore permissions to app if on local device else download and restore

Figure 6: High-level architecture of *ZapDroid*.

them. In Figure 6, we show the high-level architecture of *ZapDroid* and the interactions between its modules.

Detecting and profiling zombie apps: The detection and profiling module monitors all apps on a user's device. Apps with which the user does not interact for prolonged periods (a parameter that is set by the user; set to 1 week in our implementation) are *candidate* zombie apps. For these apps, the module then (i) tracks their resource consumption and (ii) logs the permission bits accessed by them. This allows zombie apps to be identified.

User input for quarantine: A rank ordered list of the candidate zombie apps (based on either resource consumption or permission access criteria) identified as above, is presented to the user via the front end of *ZapDroid*. The list also contains a summary of each zombie app's activity in terms of the CPU usage (in seconds), network usage (in KB/MB), battery consumption (in %), storage consumed (in MB), and the permission bits accessed by each zombie app. The user may sort the list based on any metric of her concern and tag a subset of these zombie apps to be quarantined. In addition, the user may declare these zombie apps (individually) as ones that she is likely to use in the near future ("likely to restore") or as ones that she will not be using again in the near future ("unlikely to restore").

Quarantining zombie apps: Based on the user's input as above, the quarantine module of *ZapDroid* seamlessly quarantines the chosen zombie apps. Depending on whether the zombie app is tagged as "likely to restore" or "unlikely to restore", *ZapDroid* retains it on the device (in the first case) or moves the zombie app along with its associated data to the cloud or to another form of free storage (in the second case).

No free lunch: By quarantining an app, a user essentially "removes" the app from her phone (albeit temporarily). Thus, if she stopped using the app because of the lack of some features that are later incorporated by an update, she must get to know of these new features from an external channel (e.g., a website), since her phone will not receive updates for quarantined apps. Second, if interactive apps (e.g., Skype) are quarantined, external messages using that app are not received with the current *ZapDroid* implementation, unless the user explicitly restores the app (we discuss how *ZapDroid* may be modified to overcome this later). Given this, apps are not automatically quarantined; instead, the user decides which apps are to be *virtually* removed from her smartphone. To use a previously quarantined zombie app, the user 'untags' the app from *ZapDroid*'s front end.

Restoring apps chosen for quarantine: Based on user input, the restoration module of *ZapDroid* returns the app to the exact same state that it was in, prior to quarantine. The process of restoration for "likely to restore" apps just involves re-enabling their permissions. To restore an "unlikely to restore" app, the restoration module downloads the app from where it is stored and subsequently re-enables its permissions.

Remark: *ZapDroid* exclusively resides on the user's smartphone. No changes are needed to the store to which an "unlikely to restore" app is moved. The only requirements are that `put` and `get` interfaces be implemented by the store (as is commonly the case).

Auxiliary goals: We seek to satisfy the following additional goals when designing *ZapDroid*:

- **Low overhead:** *ZapDroid* should not consume significant resources on a user's smartphone. The quarantine process should not impact device performance (e.g., offload to cloud when good WiFi connectivity is available), and restoration of apps should be quick.
- **Broad compatibility:** *ZapDroid* should be usable on *most* Android devices, if not all. In other words, it must not be limited to working on a specific vendor's device(s) or on a particular version of Android.
- **Security:** *ZapDroid* should not escalate the privileges of any application (if such escalations are allowed, potentially new unknown vulnerabilities could arise).

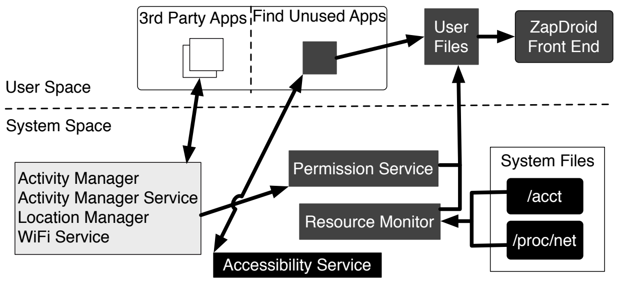
IDENTIFYING AND PROFILING ZOMBIE APPS

The first module of *ZapDroid* detects unused apps and monitors their resource consumption and permission access patterns. Any unused app that consumes resources or accesses permissions is a candidate zombie app.

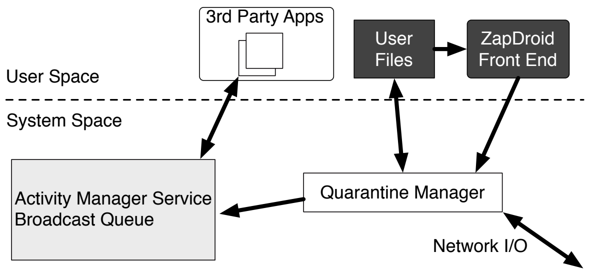
Finding unused apps: *ZapDroid* classifies third-party apps with which the user does not interact for prolonged periods (the value is set by the user via *ZapDroid*'s front end) as unused apps.

Detecting foreground apps: An app is considered as being used if it is executed in the foreground. To detect apps that run in the foreground, *ZapDroid* uses assistive technology that is part of Android;[2] since *ZapDroid* is

[2]We could have used Activity lifecycle callbacks towards identifying candidate zombie apps; instead, we simply reuse



(a) Detection



(b) Quarantine and Restoration

▨ Created ■ Used □ Modified

Figure 7: *ZapDroid*'s modules.

partly implemented in Android's Linux kernel, no user intervention is required to enable accessibility. At install time, *ZapDroid* adds an accessibility service that customizes the wake-up trigger by subscribing to the event TYPE_WINDOW_CONTENT_CHANGED. This event notifies a user space function (see Figure 7a) whenever a new window, menu, or activity is launched. Similarly, whenever a new app is brought to the foreground or when the screen is locked, the accessibility service notifies the user-space function. The user-space function records all of these events in a user file that is created upon install (this file contains a list of all third-party apps and is populated upon install). Periodically, *ZapDroid* parses the file to examine those apps that were not activated for a duration greater than the user-specified period. These apps are then categorized as the candidate zombie apps.

Monitoring network usage: *ZapDroid* uses the kernel netfilter module `qtaguid` (available since Android-3.0 Linux kernel) to track network traffic based on the UID of the owner process (which is the app's unique identifier). The output of this module is written to the file (`/proc/net/xt_qtaguid/stats`). *ZapDroid*'s Resource Manager component (see Figure 7a) checks this file every 24 hours (it uses Android's Alarm Manager to set the timers) or whenever the device is rebooted (*ZapDroid* is notified by the Broadcast Receiver) and copies the relevant content with regards to unused apps to a user file. Specifically, it records the total numbers of bytes sent and received by each unused app in that period.

Monitoring CPU usage: To monitor CPU usage, *ZapDroid* leverages Android's `cgroups` (Control Groups). *ZapDroid* reads the file `/acct/uid/<uid_of_app>/cpuacct.usage` (there is a file per UID) to determine the CPU ticks consumed by an app. The single entry in each of these files is then copied to the user file discussed earlier.

Determining battery consumption: To compute the energy consumed by an unused app, the Resource Monitor uses the linear scaling model (based on CPU ticks consumed and network traffic transferred) used in Android's Fuel Gauge [8] but at coarser time periods of 24 hours.

assistive technology that was part of `TimeUrApps`. We observe that the overheads are similar (minuscule) with both approaches (results omitted due to space constraints).

Determining storage: The Resource Monitor in *ZapDroid* calls the `du` command to measure the disk usage of an unused app. It essentially measures the space consumed by three folders (recursively): 1) `/data/data/<pkg_name>/` (the application's data on the flash), 2) `/data/data/<pkg_name>/lib` which is a soft link to the app library (the app libraries), and 3) `<External Media>/Android/<pkg_name>/` (any data on external storage such as an SD card).

Remarks: We do not use approaches that rely on `iptables` [11] or `cgroups` [19] for tracking network traffic since these approaches result in higher energy overheads without really offering any additional benefits. We also do not use Fuel Gauge or PowerTutor [49] directly since they do not account for UDP traffic transferred by some of the zombie apps (e.g., Pandora).

Tracking apps' permission access patterns: The permission access patterns of an app are not exposed to any user-level tool (or even a system-level tool unless it is enforcing the permission). Thus, *ZapDroid* requires modifications to components of Android to track the permissions accessed by zombie apps.

Tracking permissions invoked by unused apps: To track the permissions accessed, *ZapDroid* includes a new system-level service in Android called the Permission Service. It also executes a modified version of Android's Service Manager to start the Permission Service at boot time. The Permission Service does not have privileges to make modifications to any other component(s) in the OS and executes as an isolated process. Our modifications essentially cause the Android's service manager to send an asynchronous message to the Permission Service whenever it grants a permission to an app. The messaging does not block the service manager process from acting on the permission; therefore, it does not introduce any significant delays. The message consists of the UID of the app invoking the permission, the permission, and the resource that the app is trying to access (if any). Note that the messaging is essentially an IPC (inter-process communication) mechanism. It is implemented using AIDL (Android Interface Definition Language) [4]. Note also that the asynchronous messages are "one-way" (from the service manager to the Permission Service). To enable permission tracking, we had to modify many files in the Android Framework (wherever permissions were being enforced). Note that Android requires apps to create Java wrapper methods (JNI) to enable native code (C/C++) to interact with its API. These wrapper methods enable Android to enforce the permissions as with regular Java apps and do not require special consideration for native code [41].

QUARANTINING ZOMBIE APPS

Zombie apps can belong to one of two categories depending on user input: "likely to restore" (Category L) or "unlikely to restore" (Category U). For Category L, quarantine involves revoking permissions from the zombie apps and preventing them from consuming resources; the apps are however retained on the user's smartphone to ensure a quick restoration when needed. For Category U, primarily to save on storage in addition, *ZapDroid* moves the app binary and any associated data from the smartphone to remote storage. These functions of *ZapDroid* are performed by a Quarantine Manager which is implemented as an unprivileged Android system service.

User input: When a user indicates that a zombie app should be quarantined using *ZapDroid*'s front end (therein, also indicating its category), *ZapDroid*'s quarantine module is invoked. The user, using the front end, can also indicate the external storage to which a zombie app is to be moved if it belongs to Category U.

Quarantining "likely to restore" zombie apps: The Quarantine Manager of *ZapDroid* essentially kills a Category L process that is executing in the background. It also prevents other apps or processes from communicating (and thus re-initializing or waking up) this app. Note that a user-level *swipe* away of an app (i.e., switching to another app) *does not* kill the app [28].

Killing the zombie app: ZapDroid leverages Android's Activity Manager's "`am force-stop`" command to kill the chosen zombie app.

Keeping the zombie app in the inactive state: Inactive apps can potentially be activated by messages from deputy apps or cloud services like GCM (Google Cloud Message service). In an extreme case, an app may try to overcome "being force stopped" by signing up for such activation messages. To prevent such messages from reaching a zombie app killed by the Quarantine Manager, we "design and implement new" hooks to Android's Broadcast Queue (BQ) and its Activity Manger Service (AMS). The hook in BQ registers a callback from the Quarantine Manager. A function, `mCallback`, executes in the Quarantine Manager but with the privileges of BQ; the manager sends BQ a message whenever a new Category L zombie app is identified, providing the details of that app. This information is stored locally in BQ. The hook is again activated each time a new message is dequeued from BQ, to be sent out. If the message is associated with a Category L zombie app (as indicated in the stored record), the hook suppresses the message.

The hook in AMS fulfills a similar purpose. It registers a callback from the Quarantine Manager, which provides AMS with a list of zombie apps. Whenever a new activity is initiated, *intents* are used to invoke apps that are currently inactive. The hook checks the local record and suppresses intents meant for Category L zombie apps.

Quarantining "unlikely to restore" zombie apps: For a Category U zombie app, the Quarantine Manager first kills the process associated with the app (similar to Category L zombie apps). Next, it deletes the binary associated with the zombie app, and compresses and ships the data to an external storage of the user's choice. We discuss why the binary can be deleted later; based on user preferences, *ZapDroid* could also simply delete the user data (this is a clean uninstall of the zombie app).

Shipping data/binary to external storage: The external storage can be a USB stick, an alternative computer, or even cloud storage (where the user has an account). The only two APIs that we require of this storage are: 1) a PUT primitive which returns a URI (Uniform Resource Identifier) for the stored content, and 2) a GET primitive that allows the retrieval of a previously stored object based on its URI. Using the PUT primitive, the Quarantine Manager puts the user data associated with the app in external storage, and logs the URI in the previously constructed file in user space.

Why delete the app binary? The binary can be retrieved when needed from the Google Play Store. If the zombie app is restored, the data is retrieved from the external storage. If the binary is unchanged at restoration time, it can simply function with the retrieved data. A binary change is equivalent to an upgrade process, and can thus seamlessly function with the user's data (verified later). Thus, we see no point in storing the binary in the user's external store. Note that *ZapDroid* can backup the binary instead of deleting it if the user prefers to do so; this can allow the user to safeguard the possibility that the quarantined zombie app may be unavailable on the Play Store at a future time.

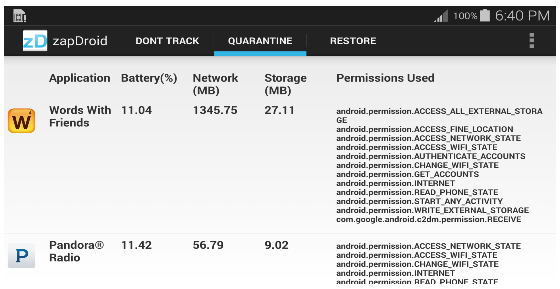
RESTORING ZOMBIE APPS

The restoration of a previously-quarantined zombie app is also handled by the Quarantine Manager. The goal is to return the quarantined app to the same state it was in (or an upgraded version if a binary is restored in the case of Category U zombie apps) prior to the quarantine.

User input: When a user seeks to restore an app, she visits *ZapDroid*'s front end, and explicitly removes the app from the list of apps to be quarantined. This automatically triggers the restoration process.

Restoring "likely to restore" zombie apps: For Category L zombie apps, the Quarantine Manager invokes the callback function to inform the hooks implemented in BQ and AMS that the zombie app to be restored must be removed from their local records. This causes BQ and AMS to forward broadcast messages and intents, respectively, to the restored app. Note that the user input will automatically launch the app.

Restoring "unlikely to restore" zombie apps: For Category U zombie apps, *ZapDroid* checks for connectivity to both the external store and the Google Play store. If the check passes, it retrieves the URI associated with the zombie app from the user file (where it had stored



	Application	Battery(%)	Network (MB)	Storage (MB)	Permissions Used
	Words With Friends	11.04	1345.75	27.11	android.permission.ACCESS_ALL_EXTERNAL_STORAGE android.permission.ACCESS_FINE_LOCATION android.permission.ACCESS_NETWORK_STATE android.permission.ACCESS_WIFI_STATE android.permission.AUTHENTICATE_ACCOUNTS android.permission.CHANGE_WIFI_STATE android.permission.GET_ACCOUNTS android.permission.INTERNET android.permission.READ_PHONE_STATE android.permission.START_ANY_ACTIVITY android.permission.WRITE_EXTERNAL_STORAGE com.google.android.c2dm.permission.RECEIVE
	Pandora® Radio	11.42	56.79	9.02	android.permission.ACCESS_NETWORK_STATE android.permission.ACCESS_WIFI_STATE android.permission.CHANGE_WIFI_STATE android.permission.INTERNET android.permission.READ_PHONE_STATE

Figure 8: ZapDroid's user interface

the information before). The URI is fetched to retrieve the associated objects. In parallel, the binary is downloaded from the Google Play Store and installed. After the install is complete, the user data is placed in the appropriate directories (this information was implicitly recorded during quarantine).

ZAPDROID'S USER INTERFACE

Fig. 8 shows a snapshot of *ZapDroid*'s user interface (UI). The candidate zombie apps, and the resources consumed and permissions accessed by each such app are listed. The user can choose to quarantine apps by simply clicking on appropriate buttons; a pop up allows the user to classify a zombie app to be either category U or L.

EVALUATION

We have a complete implementation of *ZapDroid* and evaluate it along multiple dimensions.

Validating the choice of AIDL to implement IPC: The Permission Service receives inputs from components such as the WiFi Service or the Location Manager (see Figure 7a). IPC (message passing) is used to deliver these inputs. We use Android's Binder framework, which is based on AIDL (the Android Interface Description Language) for IPC. We demonstrate that our approach is lightweight (labeled AIDL) by comparing its performance with that of the following alternative approaches: (i) using the Android's messenger framework (built on top of the Binder framework) [5] and (ii) using shared preferences [25], wherein the Permission Service is notified whenever one of the aforementioned services makes a change to an underlying shared file (stored on the SD card). In Figures 9a and 9b we depict the delays incurred in delivering the message (referred to as "timing delay"), and the energy overheads with the different approaches. We conduct these experiments on the Samsung Galaxy phones since with these phones, we were able to physically remove the battery and measure the energy using a Monsoon power meter [18]. The results demonstrate that the AIDL approach provides a three orders of magnitude advantage in terms of timing delay, and more than 60% less energy per call, compared to the alternatives. This is because, with the alternatives, there is either the overhead of using an abstraction built on top of AIDL or the overhead of using the shared media (SD card) as the medium for implementing message passing.

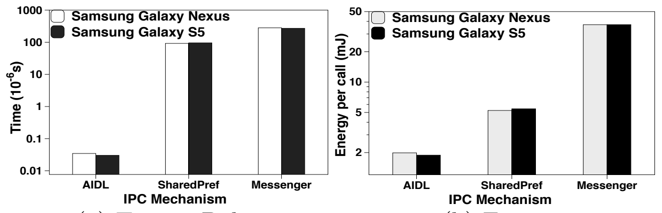
Overheads with *ZapDroid*: To quantify *ZapDroid*'s overhead we compare the timing delay and energy consumed with *ZapDroid*, built on the Android version android-4.3_r1, with that in an unmodified install of the same version. In the following, each experiment was performed 100 times and we ensured consistency.

Detection overhead: First, we examine the overheads due to notifications sent to the Permission Service. We consider permission access by unused apps, to four different services viz., the contact list, the GPS, WiFi and the notification manager. We see in Figure 10 that the increase in overhead is extremely low ($< 4\%$ with regards to energy and $< 1\%$ with regards to timing delay).

Quarantine overhead: Next, we quantify the overhead due to the modifications made to the BQ and AMS to enable quarantine. Specifically, (i) we launch an activity, *and separately* (ii) send a broadcast with the two systems. We observe from Figure 11 that the increase in timing delay and energy is $< 1\%$ for both operations. When quarantining Category U apps, in our implementation, we ship the user data either to Dropbox or to an SD card; in the former case, we ship the data when the user is on a WiFi connection and his device is plugged into a power outlet to eliminate bandwidth/energy costs.

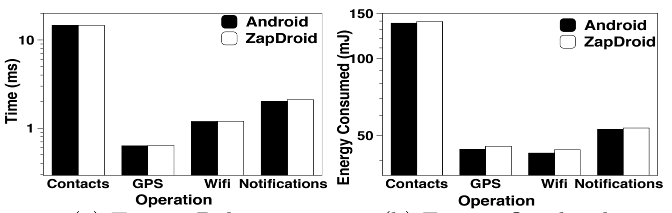
Restoration overhead: Finally, we examine the overheads due to the restoration of both Category L and Category U zombie apps. Specifically, we look at the restoration delay, and the energy expended due to restoration. The results are shown in Figures 12a and 12b. We observe that in comparison with Category U zombie apps, as one might expect, the overheads with Category L apps are minuscule; the timing delay is on the order of 5 milliseconds, and the energy overhead is about 8.7 mJ (these bars are almost not visible). With Category U apps, where this overhead is more pronounced, we consider three cases. In one case, both the binary and the user data are retrieved from the device's external SD card. In the other two cases, the binary is retrieved from the Google Play store, while the data is restored from the user's Dropbox, via WiFi and LTE, respectively. We observe that because of network transfers, in comparison with restoring the zombie app from the SD card, the overheads are approximately two orders of magnitude higher. However, even with LTE downloads, the delay incurred is less than 5 seconds with a binary plus data size of 40 MB. The energy consumed in the worst case is below 2.5 Joules (which translates to 0.007% of the battery capacity on the Samsung Galaxy S4 phone).

Verifying that restoration results in a functional app: Next, we verify that the restoration of a Category U zombie app (after a prolonged period of a month) does not hamper the app's ability to execute on a user's phone. For this experiment, we choose a random set of 50 apps (among the zombie apps from our large scale study). We installed these apps on two phones (both Samsung Galaxy S5) and had the same user data on both of them. *ZapDroid* removed all the binaries of the zombie apps from one device (these were quarantined), while we let them remain on the other device (there was no quarantine of any sort). We ran this experiment for a month and then tried to restore the zombie apps on the second



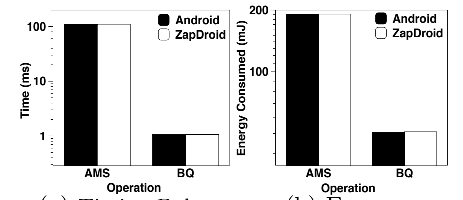
(a) Timing Delay (b) Energy

Figure 9: Comparison of IPC mechanisms.



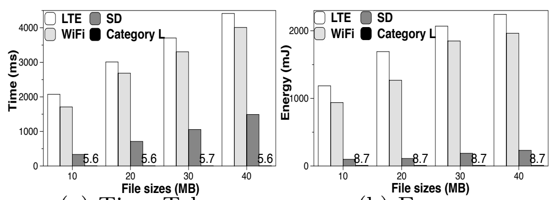
(a) Timing Delay (b) Energy Overhead

Figure 10: Permission Service overhead.



(a) Timing Delay (b) Energy

Figure 11: Quarantine overhead.



(a) Time Taken (b) Energy

Figure 12: Restoration overhead.

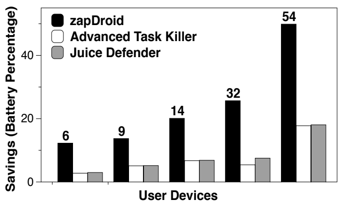


Figure 13: Comparing *ZapDroid* with other solutions.

phone (where quarantine was performed). We found that 19 of these apps were updated (later versions). On the first phone, we simply allowed these updates. On the second phone, *ZapDroid* downloaded a new binary from the Google Play store and did a fresh install. Upon launching these apps, we found that for 4 of them, the data was changed on the first phone as compared to the original data. One was due to a database upgrade and the other 3 were just additions to a file (incremental state). The apps were able to execute with the user's old information. We observed that on the second phone, the same changes were seen, and the apps were able to execute seamlessly with the user's old data. We used Android's monkey to examine if a restored app differed from the quarnatined app in terms of what was displayed on the screen (positions of objects in animations, elements in news feeds); while minor differences existed, we found that they did not affect the functionalties of the apps.

The effectiveness of *ZapDroid* in constraining zombie apps from consuming resources: We compare *ZapDroid*'s performance with that of two popular apps from the Google play store, viz., Advanced Task Killer [2] and Juice Defender [16]. These apps are designed to kill undesired background processes. For this experiment, we choose 5 user profiles from our study where there is a large variation in the number of unused apps. We select 4 *identical* Nexus 4 devices and install: (a) *ZapDroid*, (b) an unmodified version of Android with Advanced Task Killer, (c) an unmodified version of Android with Juice Defender, and (d) a version of Android with none of the above. The last case is a baseline case and we compare the performance of (a), (b), and (c) with this case. We install all the apps from the selected profiles sequentially on our 4 phone setup. We initialize all the potential zombie apps with the same state on all devices. We measure the energy saved by each system viz., (a), (b) or (c), for a 24-hour period for each profile. Figure 13 depicts the energy savings in each case. The number on top of each cluster of bars, indicates the number of zombie apps in the particular user profile. We see that *ZapDroid* saves more than 2X energy compared to the other solutions for all user profiles. This is primarily because, with Advanced Task Killer and Juice Defender (using default settings) the killed background processes restart fairly often. Then, they resume their activities as before. In fact, we notice that the savings with both Advanced Task Killer and Juice Defender are slightly lower with user profile 4 with 32 zombie apps as compared to that with user profile 3 with just 14 zombie apps. This is simply an artifact of the latter's zombie apps being more active and resource intensive upon being awake. *ZapDroid* essentially prevents these zombie apps from waking up once quarantined and thus, a much higher energy savings is achieved. Finally, we point out here that neither Advanced Task Killer nor Juice Defender helps in preventing these apps from using undesired permissions when they wake up.

Security: We verify that *ZapDroid* in no case elevated any other app's privileges. This is because it imposes an *additional*, more restrictive, constraint on the permissions granted to an app; no new permissions are granted.

DISCUSSION

Empowering applications with *ZapDroid* functions does not yield any benefits: One can conceivably empower certain apps with the features of *ZapDroid*. However, this not only still requires changes to the Android ecosphere, but also escalates the privileges of such apps which may lead to unforeseen vulnerabilities that we seek to avoid. For example, one could conceivably modify the `Launcher` app to allow it to quarantine and restore zombie apps. However, this would require the app to perform activities such as backing up the zombie app's data; such activities will require the app to be provided with root privileges (we do not provide a detailed discussion here due to space constraints). A modified version of Android that enables such privileges for a `Launcher` must therefore disallow the users from installing other third party `Launcher` apps from the market (e.g., [21]) since they would now be able to access and possibly leak users' private data.

Coping with delays experienced due to quarantine of interactive apps: Users may install interactive or social apps like Yahoo! Mail or Skype but seldom use them. However, such apps consume heavy resources for the purposes of periodically checking for updates. While allowing them to run unperturbed continuously could hurt the smartphone resources (e.g., battery), quarantining them completely is clearly not the best idea either. To address this, *ZapDroid* can be modified to reactivate such quarantined (possibly Category L) zombie apps periodically in order to enable them to check for updates. Thus restored, the app can retrieve updates, notify the user and remain active for a preset short period. It is then quarantined again. The frequency of such temporary restorations can be specified by the user; the lower this frequency, the higher the resource savings, but also the higher the latency in getting notifications.

Server side examinations of permissions are insufficient: We are concerned with the permission usage when the app executes in the background without user intervention. A server side examination will simply indicate the static permissions requested by an app rather than the dynamic usage of such permissions. Further, the determination of the frequency of usage of these permissions when the app is not actively used, requires extensive testing at the server side.

RELATED WORK

There are no papers that share our vision of identifying zombie apps, quarantining them, and restoring them if the user desires. However, there are efforts related to some of the modules that we build within *ZapDroid*.

Profiling applications: Most prior work on profiling Android applications has been either geared towards helping developers build better apps [46, 47], or detecting malicious apps [35]. There are approaches for monitoring app-specific battery consumption [48, 40]; however, they consume high energy themselves. PowerTutor [49] and the Android Fuel Gauge [8] use models to estimate the energy consumed by apps. *ZapDroid* leverages the information with regards to CPU and network provided by the Android OS. For monitoring battery consumption, we implement our own tool which functions at a coarse granularity (unlike PowerTutor). It also accounts for energy due to UDP traffic transfers (unlike Fuel Gauge).

Efforts like [45, 37] conduct studies to characterize app usage habits. However, they do not focus on infrequently used apps nor address their effective management.

Managing permissions of applications: Permission Denied [23] and XPrivacy [34] allow a superuser to change the permissions granted to apps, but the phone must be rebooted. In addition, improper user-initiated revocations can end up crashing the app or the device (due to revocation of permissions to a system app) [33]. TISSA [50] allows users to dynamically change permissions of apps. To prevent apps from crashing when they are not given access to a permission, [36] and [42] propose sending fake information to apps. Blackphone which runs on PrivatOS [10] forks a version of the Android OS and modifies it to revoke permissions towards keeping information private. In all of these efforts, the possible negative consequences from the proposed changes (e.g., the app crashing, or error pop ups [33]) has not been studied in detail. Unlike in these efforts, once a zombie app is chosen for quarantine, we freeze it by revoking *all* permissions from the zombie app and by preventing other apps from communicating with it.

Killing processes: Applications like Advanced Task Killer [2], BatteryDoctor[9], and Juice Defender[16] are meant to allow users to kill background processes associated with either services or apps to save battery. Apart from the high resources consumed by these apps themselves (they are always running and monitoring for opportunities when tasks should be killed), killing background apps using these can have other detrimental effects [31]. The killed background processes restart due to either wakeup triggers or timeouts (some start right back up [7]) and resume consuming resources and/or accessing the user's private information. Since this process of killing and restarting could happen often with these applications, even more resources may be consumed; the restarted processes will each time rebuild their state prior to being killed [32]. These functions are infrequent and thus, lightweight with *ZapDroid*.

Resource management: There are prior efforts that try to reduce resource consumption on smartphones [38, 44, 39]. However, these efforts are orthogonal to our work. We point out that Android kills background apps under low memory conditions, to reclaim memory; however, these apps can restart when the situation improves, by setting the START_STICKY bit [6]. *ZapDroid*'s goal is not just to kill apps when the memory runs low, but to curb activities of unused apps throughout.

CONCLUSIONS

Third party apps, installed but infrequently used by users, execute in the background consuming smartphone resources and/or accessing private information. We conduct an IRB-approved user study that shows the negative impact of such apps (called zombie apps). As our main contribution, we design and implement *ZapDroid*, which detects zombie apps, and based on user input, quarantines them to curb these behaviors. If needed, *ZapDroid* can later restore a quarantined app quickly and efficiently. We prototype *ZapDroid* and show that it is lightweight, and can more efficiently thwart zombie app activity as compared to state of the art solutions that target killing undesired background processes.

Acknowledgements: This work was supported by Army Research Office grant 62954CSREP and NSF NeTS grant 1320148. We thank our shepherd Dr. Markus Loechtefeld for his constructive comments which helped us significantly improve the paper.

REFERENCES

1. 108 apps per iPhone. http://fortune.com/2011/01/28/108-apps-per-iphone/.
2. Advanced Task Killer. https://play.google.com/store/apps/details?id=com.rechild.advancedtaskkiller.
3. Android: Accessibility Service. https://developer.android.com/reference/android/accessibilityservice/AccessibilityService.html.
4. Android Interface Definition Language (AIDL). http://developer.android.com/guide/components/aidl.html.
5. Android Messenger. http://developer.android.com/guide/components/bound-services.html#Messenger.
6. Android Service Lifecycle. http://developer.android.com/guide/components/services.html.
7. Android Task Killers Explained: What They Do and Why You Shouldn't Use Them. http://lifehacker.com/5650894/android-task-killers-explained-what-they-do-and-why-you-shouldnt-use-them.
8. Android's FuelGuage. https://android.googlesource.com/platform/packages/apps/Settings/+/android-4.3.1_r1/src/com/android/settings/fuelgauge/PowerUsageSummary.java.
9. Battery Doctor. https://play.google.com/store/apps/details?id=com.ijinshan.kbatterydoctor_en.
10. blackphone. https://www.blackphone.ch.
11. Block Outgoing Network Access For a Single User Using Iptables. http://www.cyberciti.biz/tips/block-outgoing-network-access-for-a-single-user-from-my-server-using-iptables.html.
12. By the Numbers: 21 Amazing Skype Statistics and Facts. http://expandedramblings.com/index.php/skype-statistics/.
13. Digital Diary: Are We Suffering From Mobile App Burnout? http://bits.blogs.nytimes.com/2013/02/15/digital-diary-are-we-suffering-from-mobile-app-burnout/?_r=0.
14. Google Cloud Messaging for Android. http://developer.android.com/google/gcm/index.html.
15. Google Play: number of downloads 2010-2013. http://www.statista.com/statistics/281106/number-of-android-app-downloads-from-google-play/.
16. Juice Defender. https://play.google.com/store/apps/details?id=com.latedroid.juicedefender.
17. Launch Checklist for Android. http://developer.android.com/distribute/tools/launch-checklist.html.
18. Monsoon Power Meter. https://www.msoon.com/LabEquipment/PowerMonitor/.
19. Network classifier cgroup. https://android.googlesource.com/kernel/common/+/android-3.10/Documentation/cgroups/net_cls.txtt.
20. Nielsen: US smartphones have an average of 41 apps installed, up from 32 last year. http://thenextweb.com/insider/2012/05/16/nielsen-us-smartphones-have-an-average-of-41-apps-installed-up-from-32-last-year/.
21. Nova Launcher Prime. https://play.google.com/store/apps/details?id=com.teslacoilsw.launcher.prime.
22. Number of available Android applications. http://www.appbrain.com/stats/number-of-android-apps.
23. Permission Denied. https://play.google.com/store/apps/details?id=com.stericson.permissions.donate.
24. Samsung Galaxy S4. http://www.samsung.com/global/microsite/galaxys4/.
25. SharedPreferences. http://developer.android.com/reference/android/content/SharedPreferences.html.
26. Smartphone users waste '£6m a year on one hit wonder apps'. http://metro.co.uk/2013/05/30/smartphone-users-waste-6m-a-year-on-one-hit-wonder-apps-3814805/.
27. strace Linux- man page. http://linux.die.net/man/1/strace.
28. What actually happens when you swipe an app out of the recent apps list? http://android.stackexchange.com/questions/19987/what-actually-happens-when-you-swipe-an-app-out-of-the-recent-apps-list.
29. What do the permissions that applications require mean? http://android.stackexchange.com/questions/38388/what-do-the-permissions-that-applications-require-mean.
30. Why do users uninstall apps? http://www.fiercedeveloper.com/special-reports/why-do-users-uninstall-apps.
31. Why RAM Boosters And Task Killers Are Bad For Your Android. http://www.makeuseof.com/tag/ram-boosters-task-killers-bad-android/.
32. Why You Shouldn't Be Using a Task Killer with Android. http://forum.xda-developers.com/showthread.php?t=678205.
33. XDA Community Support for XPrivacy. http://forum.xda-developers.com/xposed/modules/xprivacy-ultimate-android-privacy-app-t2320783/page39.
34. XPrivacy. http://www.xprivacy.eu.
35. Batyuk, L., Herpich, M., Camtepe, S. A., Raddatz, K., Schmidt, A.-D., and Albayrak, S. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications. In *Proceedings of the 2011 6th International Conference on Malicious and Unwanted Software*, MALWARE '11 (2011).

36. Beresford, A. R., Rice, A., Skehin, N., and Sohan, R. Mockdroid: Trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11 (2011).

37. Böhmer, M., Hecht, B., Schöning, J., Krüger, A., and Bauer, G. Falling asleep with angry birds, facebook and kindle: A large scale study on mobile application usage. In *International Conference on Human Computer Interaction with Mobile Devices and Services*, MobileHCI '11 (2011).

38. Calder, M., and Marina, M. Batch scheduling of recurrent applications for energy savings on mobile phones. In *Sensor Mesh and Ad Hoc Communications and Networks (SECON)* (2010).

39. Cuervo, E., Balasubramanian, A., Cho, D.-k., Wolman, A., Saroiu, S., Chandra, R., and Bahl, P. Maui: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10 (2010).

40. Ding, F., Xia, F., Zhang, W., Zhao, X., and Ma, C. Monitoring energy consumption of smartphones. In *Internet of Things (iThings/CPSCom), 2011 International Conference on and 4th International Conference on Cyber, Physical and Social Computing* (2011).

41. Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. Android permissions demystified. In *ACM Conference on Computer and Communications Security* (2011).

42. Hornyack, P., Han, S., Jung, J., Schechter, S., and Wetherall, D. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11 (2011).

43. Huang, J., Qian, F., Gerber, A., Mao, Z. M., Sen, S., and Spatscheck, O. A close examination of performance and power characteristics of 4g lte networks. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12 (2012).

44. Kemp, R., Palmer, N., Kielmann, T., and Bal, H. Energy efficient information monitoring applications on smartphones through communication offloading. In *Mobile Computing, Applications, and Services*, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer Berlin Heidelberg, 2012.

45. Parate, A., Böhmer, M., Chu, D., Ganesan, D., and Marlin, B. M. Practical prediction and prefetch for faster access to applications on mobile phones. In *ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '13 (2013).

46. Ravindranath, L., Padhye, J., Agarwal, S., Mahajan, R., Obermiller, I., and Shayandeh, S. Appinsight: Mobile app performance monitoring in the wild. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12 (2012).

47. Wei, X., Gomez, L., Neamtiu, I., and Faloutsos, M. Profiledroid: Multi-layer profiling of android applications. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12 (2012).

48. Yoon, C., Kim, D., Jung, W., Kang, C., and Cha, H. Appscope: Application energy metering framework for android smartphone using kernel activity monitoring. In *Presented as part of the 2012 USENIX Annual Technical Conference* (2012).

49. Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R. P., Mao, Z. M., and Yang, L. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10 (2010).

50. Zhou, Y., Zhang, X., Jiang, X., and Freeh, V. Taming information-stealing smartphone applications (on android). In *Trust and Trustworthy Computing*, Lecture Notes in Computer Science.
Springer Berlin Heidelberg, 2011.