Understanding and Detecting Wake Lock Misuses for Android Applications

Yepang Liu[§] Chang Xu[‡] Shing-Chi Cheung[§] Valerio Terragni[§]

[§]Dept. of Comp. Science and Engineering, The Hong Kong Univ. of Science and Technology, Hong Kong, China
[‡]State Key Lab for Novel Software Tech. and Dept. of Comp. Sci. and Tech., Nanjing University, Nanjing, China
[§]{andrewust, scc*, vterragni}@cse.ust.hk, [‡]changxu@nju.edu.cn*

1. ARTIFACT LISTING

We submit the following artifacts, including datasets and executable tools, to the Artifacts Track of FSE 2016:

1. The binaries (`.apk` files) of 44,736 Android apps that use wake locks. We collected these apps from Google Play store. The dataset (will take ∼400GB disk space) can be obtained here:

   http://sccpu2.cse.ust.hk/elite/downloadApks.html

2. A set of 31 popular and large-scale open-source Android apps that use wake locks. We provide the links to their source code repositories here:

   http://sccpu2.cse.ust.hk/elite/dataset.html

   Use this link if the above one does not work:

   https://drive.google.com/open?id=0B__1e2pwk1lWRWV0N3UzLXNCZ1U

3. A set of 55 wake lock misuse issues we found in the 31 open-source Android apps. From the issues, we observed eight common patterns of wake lock misuses, which are discussed in detail in our research paper. The 55 real issues are documented here:

   http://sccpu2.cse.ust.hk/elite/files/wakelock_issues.xlsx

   Use this link if the above one does not work:

   https://drive.google.com/open?id=0B__1e2pwk1lWVmhmR0xtNkF3VEU

4. A static analysis tool, ELITE, that can analyze Android apps to detect wake lock misuses.

2. ARTIFACTS DESCRIPTION

The first three artifacts are datasets used in our empirical study. They can be used by our community for future research. For example, the set of 55 real wake lock misuse issues can be used as a benchmark to evaluate wake lock issue detection techniques. The last artifact contains our tools and subjects for reproducing the experimental results described in Section 6 of our research paper. The tool is fully automated and can also be used by real-world developers to detect wake lock misuses in their apps. Now we further describe these artifacts along three dimensions: insightfulness, usefulness, and usability.

2.1 Insightfulness

Our study is timely and fills a gap in prior work. Wake lock is a widely-used mechanism to control the power status of Android devices. Misuses of wake locks often lead to various functional and non-functional issues, causing significant user frustrations. However, people have little understanding of how Android developers use or misuse wake lock in practice. Existing work including our prior work (references 38, 51, 58, 61 in our paper) also only studied a very limited number of wake lock issues. To bridge the gap, we conducted a large-scale empirical study on 44,736 commercial and 31 open-source Android apps. The study led to many interesting findings such as the eight common patterns of wake lock misuses. The findings not only can provide programming guidance to Android developers but also can support follow-up research on related topics. Based on these findings, we also designed a static analysis technique, ELITE, to help developers find wake lock misuses in their apps.

2.2 Usefulness

Our tool is effective and efficient. Manually analyzing complex program control and data flows to reason about whether wake locks are properly used is a tedious task. ELITE automates the process. It takes an Android app's binary or Java bytecode as input and simulates feasible execution scenarios to look for common patterns of wake lock misuses. We conducted experiments using 12 versions of five large-scale and popular Android apps. The results showed that ELITE can efficiently (analyzing each app only takes a couple of minutes) and effectively locate serious wake lock issues and outperform two existing techniques. This demonstrates the usefulness of our ELITE technique.

2.3 Usability

Our tool is fully automated and easy to use. Our ELITE tool requires no installation or configuration. It is very easy to execute. All users have to do is to prepare inputs by compiling an Android app under analysis into an `.jar` or `.apk` file and run a few scripts to invoke the tool chain. Currently, our provided scripts can only run on Linux platforms (can also run on Mac OS). Later, we will release the tool and its dependencies to public after some code refactoring and prepare scripts for other platforms.

2.4 License

Our artifacts (data, code, and runnables) are all licensed under the MIT License.

3. INSTRUCTIONS TO USE ELITE

3.1 Downloading the Tool and Subjects

First, please download the `Artifacts_Paper43.zip` file (md5 checksum = `b703fdbbd4ddd19d8c147aae54b7dc87`) at the following url (file size ∼260 MB):

http://sccpu2.cse.ust.hk/Artifacts_Paper43.zip

Use this link if the above one does not work:

https://drive.google.com/open?id=0B_ _1e2pwk1lWTGtfX3JiRHA1TlU

The zip file contains the following directories and a copy of our research paper for reference:

- The directory `tools` contains a copy of our tool implementation and scripts for running analysis tasks.
- The directory `subjects` contains the 12 experimental subjects (in `.jar` format) used in our evaluation (see Section 6 in our research paper). Running our tool on these open-source subjects can reproduce our experimental results (see Table 5 in our research paper).
- The directory `apks` contains 20 randomly selected Android apps (in `.apk` format). Unlike the above 12 subjects, these apps are not open-source.

3.2 Analyzing Android Apps (JAR Files)

ELITE can take an Android app's `.jar` file, which contains the Java bytecode of the app, as input and performs static analysis to detect wake lock misuses. Its running environment should be a 64-bit Linux machine with JRE 8.

To run ELITE on our experimental subjects (in `.jar` format), first navigate to the `tools` directory and then run the `elite_jar.sh` script from there with two arguments:

- the path to the `.jar` file for analysis
- the path to the output folder

For instance, the following command will start ELITE to analyze the `example.jar` in the `subjects` directory, and save analysis results to the sub-directory `outputs` of `tools`:

```
$    elite_jar.sh ../subjects/example.jar outputs
```

When the analysis finishes, ELITE will output three files to the `outputs` directory. The files `example.txt` and `example.err` contain console outputs from `stdout` and `stderr`, respectively. The file `example-short.txt` contains simplified analysis result for each app component c, including:

- the class name of the app component c
- whether c uses wake locks or not
- the type of wake locks if c uses wake locks
- the acquistion and releasing points of wake locks if c uses wake locks
- a list of warnings if c misuses wake locks

```
[WLA:STATUS] analyzing service=com.google.android.apps.
mytracks.services.TrackRecordingService
[WLA:USE LOCK] true
[WLA:LOCK TYPE] partial
[WLA:LOCKING SITE] <com.google.android.apps.mytracks.services.
TrackRecordingService: void onCreate()>
[WLA:LOCKING SITE] <com.google.android.apps.mytracks.services.
TrackRecordingService: void acquireWakeLock()>
[WLA:RELEASING SITE] <com.google.android.apps.mytracks.
services.TrackRecordingService: void onDestroy()>
...
===policy violation===
component class: com.google.android.apps.mytracks.services.
TrackRecordingService
wake lock should be released at: void endCurrentTrack()
example sequence: <com.google.android.apps.mytracks.services.
TrackRecordingService: void onCreate()><com.google.android.
apps.mytracks.services.TrackRecordingService: int
onStartCommand(android.content.Intent,int,int)><com.google.
android.apps.mytracks.services.TrackRecordingService: void
endCurrentTrack()><com.google.android.apps.mytracks.services.
TrackRecordingService: void onStatusChanged(java.lang.String,
int,android.os.Bundle)><com.google.android.apps.mytracks.
services.TrackRecordingService: void onStatusChanged(java.lang
.String,int,android.os.Bundle)>
```

Listing 1: Example output when analyzing MyTracks (revision f2b4b968df)

Listing 1 gives example analysis result of running ELITE on our experimental subject MyTracks (revision f2b4b968df). In the example, ELITE reports that the service component `TrackRecordingService` uses a partial wake lock and the wake lock is acquired and released in the `onCreate()` and `onDestroy()` lifecycle event handlers of the component, respectively. After analysis, ELITE detects that the component could suffer from unnecessary wakeup issues and reports warnings accordingly. To ease issue diagnosis, ELITE also provides the method call sequences leading to the detected issue and the program points where wake locks should be released (i.e., `endCurrentTrack()` in the example).

3.3 Analyzing Android Apps (APK Files)

ELITE can also take an Android app's `.apk` file, which contains the Dalvik bytecode of the app, as input for analysis. The process of running ELITE on an `.apk` file is similar to that of running ELITE on an `.jar` file. The only difference is to use another script `elite_apk.sh`.

We prepared 20 `.apk` files in the `apks` directory for testing ELITE. The file names follow the format "package_name.apk", where the package name is the app's unique ID declared in the `AndroidManifest.xml` file.[1] If you want to prepare your own `.apk` files for experiments, please name them according to the format. ELITE currently relies on file naming to recognize an app's package name for analysis.

3.4 Analyzing Android Apps in Batch Mode

To ease the artifact evaluation, we also provided scripts to run ELITE to analyze a directory of `.jar` or `.apk` files.

To run ELITE in batch mode, please navigate to the `tools` directory and then run the scripts `elite_jar_batch.sh` and `elite_apk_batch.sh`. The scripts take four arguments:

- the path to the directory of the `.jar` files (or `.apk` files respectively) for analysis
- the path to the output folder
- the maximum number of concurrent jobs
- the total files to be analyzed

For instance, running the following command will start ELITE to analyze 20 `.apk` files in the `apks` directory with 6 concurrent jobs running at the same time and output the analysis results to the `outputs` directory.

```
$    elite_apk_batch.sh ../apks outputs 6 20
```

[1]https://developer.android.com/guide/topics/manifest/manifest-element.html