Chiromancer: A Tool for Boosting Android Application Performance

Samit Anwer Aniya Aggarwal Rahul Purandare Vinayak Naik

Indraprastha Institute of Information Technology, Delhi
New Delhi - 110020, India
{samit1274,aniya1234,purandare,naik}@iiitd.ac.in

ABSTRACT

Each Android application runs in its own virtual machine, with its own Linux user account and corresponding permissions. Although this ensures that permissions are given as per each application's requirements, each permission itself is still broad enough to possible exploitation. Such an exploitation may result in over consumption of phone's resources, in terms of processing, battery, and communication bandwidth. In this paper, we propose a tool, called *Chiromancer*, for the developers and phone users to control application's permissions at a fine granularity and to tune the application's resource consumption to their satisfaction. The framework is based on static code analysis and code injection. It takes in compiled code and so does not require access to source code of the application. As a case study, we passed publicly available applications from Google Play through *Chiromancer* to fine tune their performance. We compared energy and data consumed by these applications before and after the code injection to corroborate our claims of improvement in performance. We observed substantial improvement.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Performance

Keywords

Android, Performance, Static Analysis, Code Injection, Mobile, Soot

1. INTRODUCTION

Android has about 75% of the market share with a growth rate of 91.5% in the last year [10]. Undoubtedly, Android is the most pervasive mobile OS. With increasing rate of cybercrime, people are more concerned about malwares and privacy issues. However, performance issues are equally important as they might impact data and battery consumption of an application. This motivates us to target Android applications (apps) for proposing performance optimizations.

In this paper, we present *Chiromancer*, a prototype that takes Application Package (APK) of apps and performs static analysis to uncover performance issues. After detection, the tool transforms the APK based on the inputs provided by the user so that the app gets tuned according to the user's needs rather than the developer's whims. Such modifications may not always be meaningful since the user is completely code agnostic. However, the user will be told about the implications of the modifications a priori.

Our approach relies on the fact that the user understands her needs better than the developer. We assume that the user must have an idea about the kind of application she is going to install, which is generally true or can be known by reading the app's features on the store. This way, the user can decide certain aspects of the application. For instance, assume that the app is a game like Chess. Time is not a major factor in the game, so a user can choose to have a *PARTIAL_WAKE_LOCK* [5] instead of a *FULL_WAKE_LOCK* [5]. This would allow the screen to go off after a while thereby saving energy. In contrast, for a car racing game it would not be wise to keep a *PARTIAL_WAKE_LOCK*.

We claim that a static analysis technique (like Soot [6]) is better than using AspectJ [11] because, firstly, AspectJ cannot detect the presence of method calls in the code statically. Hence, it cannot be used to skip some portion of the code based on whether a specific method call is present in the rest of the code, whereas, Soot can do this easily. Secondly, it is not possible to intercept assignment statements using AspectJ hence it cannot be used to perform sophisticated analysis like data flow analysis. Thirdly, using AspectJ, one can only go "around" the code that sends an SMS and cannot remove it from the app's binary. These limitations of AspectJ diverted our attention to Soot.

The purpose of our work is neither to hack the app (as the modified APK would need to be re-signed) nor to infringe copyrights. The work is not targeted to gain monetary benefits, rather it is aimed to show that such performance oriented optimizations are possible by leveraging code injection, without requiring source code of apps. The tool can be used by the general public or the app developers to detect and overcome performance issues. In this work, we also propose a framework which can be used by developers to customize the optimizations provided by Chiromancer and extend them

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

MOBILESoft'14, June 2–3, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2878-4/14/06...$15.00
http://dx.doi.org/10.1145/2593902.2593918

in various ways. As per our knowledge, there has not been much work done to detect and rectify performance issues in code extracted from APKs. In this context, our work elicits a novel approach to detect and deal with performance issues.

2. CHIROMANCER FRAMEWORK

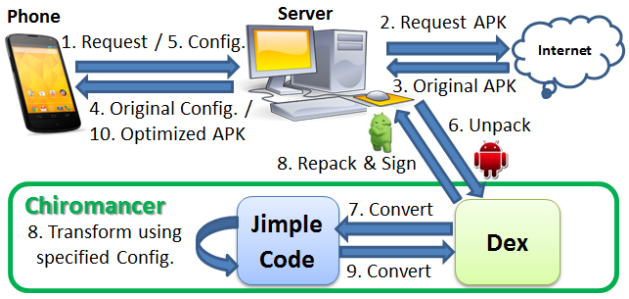

Figure 1: Chiromancer Architecture

The architecture of Chiromancer is shown in Figure 1. The user places application request (step 1) to our server. The requested app's APK is then fetched by our server (step 2 and 3) from the Internet and statically analyzed to detect probable performance issues. The performance related configuration values originally present in the application are reported back to the user (step 4). The user can then specify the configuration changes she wishes to apply to the app using the GUI (step 5). Chiromancer, then modifies the requested APK based on these inputs and re-signs the modified APK (step 6, 7 and 8 and 9). This APK can be hosted on a server and the link to it can be sent to the person who requested the optimized version (step 10).

The work horse of Chiromancer is Soot's [6] modification known as Dexpler [9], which takes an APK as input, unpacks it, finds the Dalvik bytecode (.dex) and converts it directly into Jimple. Soot's phases run one by one. The *Transformation phase* executes first. In this phase the application's Jimple code is analyzed to detect method signatures of interest and the relevant Jimple statements are inserted or parameters modified according to the input configuration. Next, the *Optimization phase* optimizes the code and finally the *Annotation phase* performs built-in analysis. The resultant Jimple code is now performance optimized.

We have built a custom framework on top of Soot that provides three levels of abstraction and thus provides an easy way to develop new performance optimizations or extend the existing ones. The outermost level provides an interactive GUI-based functionality which enables users to register themselves and get their desired APKs optimized without requiring any knowledge about the internal code. The next two levels modularize the framework and allow the developers to build custom optimizations on top of our API. The multiple abstraction levels relieve the developer from worrying about the complicated code structures that are required to be constructed while instrumenting using Soot.

The middle level of the framework captures high level functionalities that aim to improve domain specific performance. This level comprises of four domain specific modules - GPS Update Frequency Optimizer (GUFO), Wake Lock Optimizer (WLO), Advertisement Remover (AR), and SMS to Premium Number Eliminator (SPNE). These modules correspond to the four performance issues which are discussed later in section 3. GUFO module has one of the methods declared as - *void modifyTimeDist(long minTime, float minDist)*. Here, *minTime* and *minDist* refer to the minimum time interval (in milli seconds) and the minimum distance interval (in meters) respectively at which the location updates are requested. One can set the desired location update frequency by just passing desired values as arguments to the function call. Likewise, other modules also have several other relevant method declarations. This middle layer would expand accommodating support for more optimizations in the future.

The innermost level of abstraction is typically meant for the developers who wish to develop new performance-oriented optimization modules on top of our API and Soot's API. This level contains the generic core functionalities which can be used across several optimization modules residing at the middle level of framework. One such functionality used by the GUFO and WLO is - *void mod_Param*($Unit\ unit$, $int\ param\_idx, < T > param\_val$), where the first argument *unit* of type *Unit* [6] is the method whose parameter value at index *param_idx* will be replaced by the value passed in *param_val* having type *T*.

3. OPTIMIZATIONS

In this section, we describe the various issues and propose solutions to tackle them.

Issue 1. Advertisements in applications

Most apps these days have advertisement (ad) banners. According to AppBrain [4], Admobs [1] is the most famous API for in-app advertising. These banners fetch ads from the Internet and consume a significant portion of the total network data consumed by the app. Hence, permission to access the Internet can also be misused by the developers.

```
$r18 = $r0.<com.lul.accelerometer.AccelerometerMonitorActivity:
                                    com.google.ads.AdView adView>;
$z0 = 1;
if $z0 == 1 goto label0;

virtualinvoke $r18.<com.google.ads.AdView:
                    void loadAd(com.google.ads.AdRequest)>(r31);

label0:
  nop;
  r32 = new com.lul.accelerometer.SignalChart;
```

Figure 2: Advertisement application Jimple code after injection

Solution: In order to avoid consumption of data by advertisements we use the idea of skipping execution of the *loadAd(AdRequest new AdRequest())* by injecting appropriate code. Figure 2 shows the Jimple code after injection of code to skip an advertisement load call. The users, prior to installation, can decide whether they wish to see any advertisements and can then proceed to install the application. In contrast, the authors of [8] propose two use cases - *Adremover* and *FineGPolicy*. *AdRemover* assumes that developers place potentially dangerous code in try/catch blocks, however, this might not always be true. *FineGPolicy* causes invocation of method stubs instead of actual methods for those methods for which permission has not been granted. This increases the overhead. Furthermore, in [8] the authors instrument the app's bytecode directly on the smartphone. This would drain the battery very quickly. Our technique can be extended to work for other advertisement libraries also by adding their respective load advertisement method signatures.

Issue 2. SMS to premium rate numbers

A very common issue that incurs cost is sending SMSs to premium rate numbers. Many applications ask the permission for sending SMSs. Unfortunately, there is no restriction built in to the security model of Android that checks whether the target number is a premium rate number or not.

Solution: To avoid exorbitant expenditure we inject application code to skip execution of method *sendTextMessage(String destinationAddress, String scAddress, String text, PendingIntent sentIntent, PendingIntent deliveryIntent)* if the recipient's number is a premium rate number, just like the authors have shown in [7]. The control would skip over the code that sends the message, only if the target number begins with "0900", and resume execution from a *nop* statement. We further extend this implementation to take care of a situation where an SMS to a premium number is sent and a response is awaited as a reply. This might happen when some other task depends on the content that is sent back as a response. This entails implementing a Flow Insensitive Analysis that checks whether the code has a *createFromPdu (byte[] pdu)* method which converts a Protocol Data Unit (PDU) to an *SmsMessage* [5] object. This confirms that the message contents will be used further. We use Flow Insensitive Analysis because Android apps have multiple entry points and manifest a number of different flow sequences without any specific ordering of method calls. Hence, it is difficult to accurately predict whether the *createFromPdu ()* method comes before or after the *sendTextMessage()* method call. It is evident that such conditional skipping of method calls would not have been possible by using AspectJ because of its inability to discover a call to *createFromPdu ()* when it detects a *sendSms() Join Point* [11].

Issue 3. GPS location update frequency is too high

Global Positioning System, now common in smartphones, used to accurately locate a user, is an accomplice in draining battery. GPS requires communication with three to four satellites to accurately locate a device. The antenna needs to be powered all the time during the communication period. This prevents the phone from going into a sleep state. Matters get even worse when the location updates are required very frequently with short time gap and short difference in distance. Short location update intervals are often unnecessary and cause significant battery drain.

Solution: We let the users decide the parameters to be passed to the *requestLocationUpdates(String provider, long minTime, float minDistance, LocationListener listener)* method. Users know better whether they will be travelling on foot, by train, or by car. Therefore it becomes necessary to give the users the freedom to decide the location update intervals. The users can input the frequency at which they want the application to make location updates. This value is replaced in the parameter of the *requestLocationUpdate()*.

Issue 4. Wake lock is acquired for long time

The wakelock mechanism informs an application about the duration for which the device should keep its screen, CPU or keyboard active. The screen consumes maximum battery in Android phones. Careless use of this API can drain the battery quickly. Various wakelock levels along with their corresponding constant values are described at [5].

Solution: We allow the users to decide which wakelock they want the application to acquire. Then we replace the parameter of the *PowerManager.WakeLock newWakeLock()* by the one supplied by the user. For instance, a user can replace a *FULL_WAKE_LOCK* by *PARTIAL_WAKE_LOCK* [5] to save energy.

4. STRENGTHS AND WEAKNESSES

The strengths of our approach are as follows. Chiromancer detects method calls by an exact match of their method signatures. Consequently, it provides fine grained control over the solution for the issues and can adapt to different Android API levels. For instance, *sendTextMessage()* has a number of different parameter combinations for performing the same task. But, since we detect these methods by an exact match of their parameters we can make signature specific optimizations. Thus we get very accurate results (in terms of false positives and false negatives). In addition, the use of Dexpler avoids an intermediate conversion to Java Bytecode and hence, speeds up the overall process. Finally, our approach is intuitively scalable to large code bases as it relies on Soot for most of its functionality.

Despite providing the end-users with the paraphernalia to improve the performance of the application, Chiromancer suffers from a few drawbacks. Firstly, the tool does not run on the phone. The APK needs to be transferred to a PC for modification. This is an advantage in a way because Soot can sometimes be slow and require a lot of memory. Hence, unlike [8], we avoid direct instrumentation on the phone. Secondly, any configuration change would require a user to request the server for another APK instrumented appropriately with new parameters. Thirdly, the modified APKs need to be re-signed before they can be deployed on smartphone, and this requires the users to trust our APKs. Moreover, re-signing would stop automatic app updates.

5. EVALUATION

In this section we elicit the improvements that Chiromancer yields using three third party applications.

5.1 Testbed

The applications - `Accelerometer Monitor`, `Swing Ball`, and `GPShake Lite` were chosen because they had advertisements, dim wake lock [5], and very frequent GPS location updates respectively. We performed our experiments with these three applications on two Android phones - a) Sony Xperia P LT22i (dual-core, 1 GHz Cortex-A9 processor, 1 GB RAM, Android version 4.1.2), b) HTC Explorer (single-core, 600 MHz Cortex A5 processor, 512 MB RAM, Android version 2.3). For `Accelerometer Monitor` application, which has an advertisement, we are concerned only with the network data consumption, whereas for `Swing Ball` and `GPShake Lite`, only energy consumption is considered.

Xperia P had screen brightness set to "Adapt to lighting conditions" and the display timeout set to 1 minute. On Explorer, the brightness was set to 50% and the timeout was set to 10 minutes. All the readings for `Swing Ball` application were taken in the same light conditions.

5.2 Methodology of Collecting Results

We ran two versions (unoptimized and optimized) of the same application on the phones and collected energy and data consumption readings (whichever appropriate). The optimized (OPT) and unoptimized (U-OPT) versions of the application were both run ten times for a duration T = 180 seconds each. The energy and data consumption were

Table 1: Evaluation Results

Application	Version	Metric	Sony	HTC
Accelerometer Monitor	U-OPT	D_{avg}	20.4	17.99
		R	10	10
		S.D.	2.57	0.43
	OPT	D_{avg}	1.67	0.53
		R	10	10
		S.D.	0.97	0
Swing Ball	U-OPT	E_{avg}	114.84	176.02
		R	10	10
		S.D.	3.31	4.01
	OPT	E_{avg}	60.88	73.5
		R	10	10
		S.D.	3.13	1.6
GPShake Lite	U-OPT	E_{avg}	1.98	1.42
		R	9	9
		S.D.	0.28	0.14
	OPT	E_{avg}	0.53	1.02
		R	9	9
		S.D.	0.15	0.05

Here:
U-OPT : Un-optimized and OPT : Optimized
E_{avg} : Average energy consumed by the application in Joule where the average is over R readings of E
E : Total energy consumed by application in Joule in time T
D_{avg} : Average network data consumed by the application in KB where the average is over R readings of D
D : Total network data consumed by application in KB in time T
R : Number of readings over which E_{avg} and D_{avg} are calculated
S.D. : Standard Deviation of the R readings

measured by using `PowerTutor` [3] and `Traffic Monitor` [3] respectively. Table 1 tabulates the data and energy consumption statistics obtained from both the phones for the three applications.

`Accelerometer Monitor` [2] application has an ad from Google's AdMob [1], which forms a significant portion of the total network data consumption. Our tool generates an optimized version of this application by skipping the *loadAd()* method calls. The statistics in Table 1 show that data consumption for this application reduced by a factor of 12.2 for Xperia P and by a factor of 33.94 for Explorer.

Similarly, `Swing Ball` application [2] was used to show improvements achieved by wakelock manipulation. This application originally had *SCREEN_DIM_WAKE_LOCK* [5] which was changed to *PARTIAL_WAKE_LOCK* [5] after optimization. The total energy consumed, E is recorded as $E = E_{CPU} + E_{LCD} + E_{3G}$, where E_{CPU}, E_{3G} and E_{LCD} denote the energy consumed by CPU, 3G service usage and device screen respectively in time T. Statistics from Table 1 clearly show that energy consumption for `Swing Ball` application reduced by a factor of 1.88 in Xperia P and by a factor of 2.39 in Explorer.

Finally, `GPShake Lite` application [2] was used to show the improvement achieved after reducing the location update frequency to 200 meters or 10 sec from 20 meters or 0 sec respectively. The total energy consumed, E is given by $E = E_{CPU} + E_{3G}$, where E_{CPU} and E_{3G} represent the energy consumed by CPU and 3G service respectively for time T. The results in Table 1 show a decrease in energy consumption by a factor of 3.7 on Xperia P and by a factor of 1.39 on Explorer. GPS consumes considerably more time and energy for the first location fix. Hence, the first reading, being an outlier, has been ignored and the average has been taken over the remaining nine readings. The improvements offered by Chiromancer are evident from these experiments.

6. CONCLUSION AND FUTURE WORK

We presented a tool that performs performance optimizations on Android applications using static code analysis. The tool provides an extensible framework that allows the developers to tune parameters and build new optimizations. We are working on the API to add more functionality and make applications interactive so that they can accept parameters from the user at runtime. We will also add support for more sophisticated data flow analysis based optimizations in the future. Few more performance issues that we wish to overcome are as follows. a) Allow apps to download content only in the vicinity of a Wifi zone. b) Disallow heavy work on the Main/UI thread to avoid un-responsiveness. c) Prompt user to suspend the apps that make use of Internet, if the battery is low. With the existing features and those that will be added in future, we hope that Chiromancer helps users and developers to improve their app's performance.

7. ACKNOWLEDGMENTS

We would like to thank Dr. Eric Bodden for sharing his expertise and insights throughout the course of this work.

8. REFERENCES

[1] Admob - monetize and promote your mobile apps with ads - google ads. `http://www.google.co.in/ads/admob/`.
[2] Android Apps, Download APK. `http://www.appsapk.com/`.
[3] Google Play. `https://play.google.com/store`.
[4] Number of available Android applications - AppBrain. `http://www.appbrain.com/stats/number-of-android-apps`.
[5] Package Index - Android Developers. `http://developer.android.com/reference/packages.html`.
[6] Soot: a Java Optimization Framework. `http://www.sable.mcgill.ca/soot/`.
[7] S. Arzt, S. Rasthofer, and E. Bodden. Instrumenting Android and Java Applications as Easy as abc. In *RV*, pages 364–381, 2013.
[8] A. Bartel, J. Klein, M. Monperrus, K. Allix, and Y. L. Traon. Improving Privacy on Android Smartphones Through In-Vivo bytecode instrumentation. *CoRR*, abs/1208.4536, 2012.
[9] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon. Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot. In *Proceedings of the International Workshop on the State Of the Art in Java Program Analysis (SOAP'2012)*, 2012.
[10] I. D. Corporation. Worldwide Quarterly Mobile Phone Tracker. `http://www.idc.com/tracker/showproductinfo.jsp?prod_id=37`.
[11] Eclipse. The AspectJ Project. `https://www.eclipse.org/aspectj/`.