Do the Stars Align? Multidimensional Analysis of Android's Layered Architecture

Victor Guana, Fabio Rocha, Abram Hindle, Eleni Stroulia
Department of Computing Science
University of Alberta
Edmonton, Canada
{*guana, fabiorocha, abram.hindle, stroulia*}*@cs.ualberta.ca*

***Abstract*—In this paper we mine the *Android bug tracker repository* and study the characteristics of the architectural layers of the Android system. We have identified the locality of the Android bugs in the architectural layers of the its infrastructure, and analysed the bug lifetime patterns in each one of them. Additionally, we mined the bug tracker reporters and classified them according to its social centrality in the Android bug tracker community. We report three interesting findings, firstly while some architectural layers have a diverse interaction of people, attracting not only non-central reporters but highly important ones, other layers are mostly captivating for peripheral actors. Second, we exposed that even the bug lifetime is similar across the architectural layers, some of them have higher bug density and differential percentages of unsolved bugs. Finally, comparing the popularity distribution between layers, we have identified one particular layer that is more important to developers and users alike.**

***Keywords*-bug tracker; architectural layer; lifetime; social centrality**

I. Motivation and Background

The ability to mine software repositories and specifically bug tracker systems provides software architects and developers the facility to discover where they have been investing time fixing problematic issues of a system. As the bug accountability capabilities for developers increase, so does the ability to make decisions about team distribution or resource investment for sensible software architecture structures. Historical data provided in the bug tracker repositories can unveil unexpected behaviors aligned to concrete parts of the architecture where bug characteristics like lifetime, bug density, or user's interest differ.

Some authors use system bug reports as a main source of information for layered architectural analysis. For instance, HATARI [1] integrates structural information from source code and bug repository entries to measure how components are affected by bugs and which bugs affect more components. Kumar et al. [2] look into the manifestation of failures in different modules of Android and their characteristics. We use similar approaches and extend current proposals by using social network and community interest metrics.

The 2012 Mining Challenge has provided a subset of the Android bug tracker repository for exploration and analysis [3]. We have leveraged this information, together with the Android layered architecture in order to explore how the architectural locality of a bug relates to its lifetime, popularity, and authorship. In our paper we investigate 4 questions in particular:

1) *Is there any architectural layer with higher bug density?*
2) *Is the bug lifetime affected by its location within the system architecture?*
3) *Is there any architectural layer where central bug reporters focus their attention?*
4) *Do participants of the bug tracking community follow any architectural location in particular?*

Our study identifies architectural hotspots where our conceptual framework could be used to develop social-centric software bug triage and development strategies.

II. Android Layered Architecture

The Android Operating System (OS) is a set of software services specially ported for mobile devices. Its architecture is based on five layers where different functionalities and behaviors take place: applications, framework, libraries, runtime, and linux kernel. The application layer consists of individual applications such as email clients, browsers, or games that extend the OS functionality. The library layer configures a set of C/C++ packages used by the application framework to manage the screen rendering, device security, applications persistence, among others. The runtime layer is composed by the Dalvik virtual machine and Android core libraries that specify the applications execution environment inside the OS. At the bottom of the layers of abstraction, a customized linux kernel is used in order to provide the low level OS capabilities such as memory management and process scheduling. This architecture is widely known and has been summarized in [2]. Given that bugs related with the applications are reported in independent bug tracking systems, we have studied the bugs related with the four last layers omitting the application level.

III. Input Data

The bug tracker log provided by the MSR challenge contains 20169 bug entries, committed by 11617 different reporters. For our particular interest, the structure of the bug entries involves the bug id, open and close date, description, reporter id, and owner. Additionally, each bug has an

978-1-4673-1761-0/12/$31.00 © 2012 IEEE

optional set of comments (including its author, title, and submission date). A stars field is used in order to represent the number of people following a bug (getting notifications if the bug report presents any activity). Moreover, the log contains 67730 comments, limited to 25 in total per bug, with an average of 3.35 and a deviation of 5.33. Bug comments and descriptions are particularly rich; on average, while a bug description has 113.1 words, its comments have 139.4 (with word deviations of 130.65, and 338.4 respectively). The recorded bugs were submitted between November of 2007 and December 2011, providing a partial bug history of 50 months.

IV. Methodology

We have investigated the bug locality within the Android architecture presented in Section II and their characteristics. We have used a token-set schema that allows performing set operations for counting and filtering properties. We supported the reporter social analysis by creating relational tables and running SQL exploratory queries. In this section we discuss the exploration framework developed to answer the four questions mentioned in Section I.

A. Step 1: Bug Layer Classification

First, we have categorized the bugs according to topic words related to each one of the architectural layers, we have selected specific set of words and its occurrence inside the bug title, description, and comments as classification variables of the bug. We have performed this step given that the `<component>` field is empty in almost all the bug reports. Using our classification methodology, if at least 10% of the bug total word count is related with a layer topic set, it is labelled with the layer name. The topic words per layer are based on the Android architectural documents[1]:

Framework: *Activity Manager, Package Manager, Telephony Manager, Window Manager, Content Provider, Location Manager, View System, Notification Manager.*

Libraries: *Surface Manager, Media Framework, SQLite, OpenGL, FreeType, WebKit, SGL, SSL, libc.*

Runtime: *Core, Dalvik, Virtual Machine.*

Kernel: *Display Driver, Camera Driver, Flash Memory Driver, Binder, IPC, Keypad Driver, WiFi Driver Audio Driver, Power Management.*

B. Step 2: Bug Lifetime

Focusing on the bugs that have been associated with some architectural layer, we exclude those that have not yet been closed. Then we calculate the lifetime of each bug as the number of days lapsed from the date it was created to the day it was closed. We have taken the life window of each bug as the period between its the open and closed date.

[1] http://developer.android.com/guide/basics/what-is-android.html

C. Step 3: Bug Reporters Social Network Analysis

We studied the social importance of the Android layers by analysing the social centrality of their bug reporters and how popular each layer is for the general community within the 50 months study window. In order to analyse the centrality of a bug reporter we have used the reporters email addresses as their unique id given each bug `<reportedBy>` field. Using a social network analysis approach we have characterized the importance of a reporter as its social centrality (sum of the reporter indegree and outdegree). We mined each reporter social outdegree as the number of bugs they reported, plus the number of comments they have posted in other existing bugs. The reporter indegree has been calculated as the sum of the number of comments that his bugs have received.

D. Step 4: Bug Staring Analysis

The `<stars>` field of each bug specifies the number of people following the bug activity. Thus, as a proxy measure of the amount of interest each layer is receiving from the community, we count the number of stars associated with the bugs at each layer (as identified in step # 1 above).

V. Results

After the filtering process our methodology classified close to 43% of the reported bugs among the four architectural layers. 57% of the bugs were not classified inside the layers under study. Most of the non-classified bugs belong to particular applications in the applications layer or have been posted as system open questions. The results are summarized in Table I. We have sampled 40 of the classified bugs and manually validated its location within the architecture, we found that 31 of them were correctly classified giving our methodology an accuracy of 77.5%.

Table I
Bug Layer Classification

Layer	Total	Bug Tracker %
Framework	**4756**	**23.58**
Libraries	744	3.68
Runtime	612	3.03
Kernel	2485	12.32
Not Classified	**11572**	**57.37**

1. Is there any architectural location with higher bug density? Our classification method exposed a higher bug concentration in the framework layer (Table I). We believe that this is because the bug tracker community is mainly composed by application developers which use the framework managers in order to support the application control over the mobile device capabilities. Furthermore, the study shows that the kernel layer also presents a considerable concentration of bug reports. A manual inspection showed a community trend on developing platform drivers to customize the OS to experimental hardware.

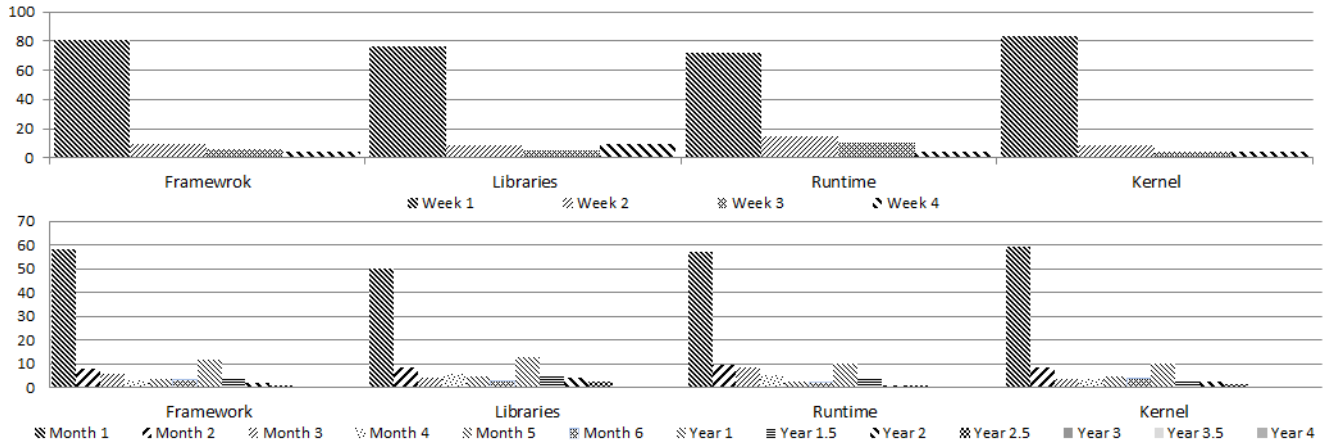

Figure 1. Bug Lifetime - Android Layer Comparison (Month 1, and Years 1 to 4 windows)

As a first step we localized the proportion of closed bugs in each one of the layers. We found that in the framework layer around 32% of the bugs were closed in the time window under study. Moreover, 46% of the bugs in the runtime layer were closed, making it the one with more closed bugs in the architecture. We focused on bugs with an interesting lifecycle (other than creation-initial inspection-closing) and we excluded bugs with the labels *Duplicate, NeedsInfo, Spam, and Unreproducible* in their `<status>` field, we named this subset the discardable bug set. Table II summarizes the closed bugs per layer.

Table II
CLOSED BUGS IN ARCHITECTURAL LAYERS

Layer	Closed	Layer %	Non-Discardable %
Framework	**1524**	32.04	20.43
Libraries	203	27.28	18.32
Runtime	283	**46.24**	**37.18**
Kernel	637	25.63	12.51

2. Is the bug lifetime affected by its location within the system architecture? The lifetime across layers is rather similar. Figure 1 portrays the bugs lifetime in two different time windows: first month (up) and a 4 years (bottom). In Figure 1, the *y* axis shows the percentage of bugs closed from the total reported in 50 months study window.

There is a clear pattern where most of the bugs are closed in their first month of existence. In the libraries layer, the initial (first month) close proportion is smaller in comparison with the other ones (50% versus almost 60% in the rest of the cases). Similarly, during the first year, the closing proportion decreases gradually except for the libraries layer where, in months four and five, the proportions are 5% higher than in the other layers. We suspect this behavior is because most of the components in this layer (e.g. SQLite, WebKit, OpenGL-ES) are maintained by third-party teams; it takes around 2 months to redirect, assign, and close them.

Other observations expose that 89% to 93% of bugs are solved in their first year of existence. However, the framework and kernel layers present long lasting bug densities after the year 2.5. In these layers between 1.4% and 1.9% of the bugs are solved late in terms of our study window (after 2.5 years), a manual inspection revealed that most of them are related with the OS portability, and were reported and closed by an OS version release date. In this time frame we observed features such as USB drivers, multimedia codecs, and network security that were requested and completely developed. The first month window confirms the proportion of the discardable bugs. Here we can observe that most of the bugs solved in the first month are closed in the first week, suggesting that most of them belong to each layer discardable bug set. In particular 49% of the closed bugs in the framework layer are discardable and were closed during their first two weeks of existence.

3. Is there any architectural layer where central bug reporters focus their attention? In terms of the reporters centrality in the bug tracker network, we found that while committers in the framework layer have on average a higher centrality, reporters on the library belong primarily to the network periphery (Table V). Additionally, using Violin Plots [4] we can observe in Figure 2 that the density of the kernel and framework layers have a great diversity in terms of reporters centrality. Especially in the framework layer, a concentration of low, medium and high central reporters can be observed in a left skewed normal distribution. On the other hand the bugs in the library and runtime layers are not reported by the bug tracker network top central reporters. We believe this happens because the network is particularly popular for high level application developers, causing a great interest in the framework components used to enhance the top level applications experience. As a consequence the

reporters of the framework layer bugs are more involved feedback discussions that increase their centrality. Library and runtime reporters act in the network periphery inside more controlled discussion clusters. As an example of our social network analysis, the most central contributor in the network is Jean-Baptiste Queru, Android's open source code leader (6190 incident ties).

Table III
LAYER REPORTER CENTRALITY SUMMARY STATISTICS

	Rep. Centrality (AVG)	Rep. Centrality (STD)
Framework	71.17	258.35
Libraries	32.25	56.54
Runtime	37.36	78.68
Kernel	49.30	92.80

4. Do participants of the bug tracking community follow any architectural location in particular? Using the distribution of bugs followers provided by the stars field we characterized the interest of the community in each layer. While the reporter centrality limits the popularity analysis to the formally involved nodes of the bug tracker network (commenter and reporters), the stars provide a more general indicator given anyone can register as a bug follower. Figure 3 and Table IV, expose that in addition to the framework layer, the kernel is also popular for the follower community based on their star concentration. Upon log inspection, this happens because general users follow the OS hardware compatibility issues as well. Those issues are classified in the kernel layer mostly, where the problems related to drivers and software to hardware adaptation are spotted. The significance of this and the previous question relies on how development and management teams could identify architectural zones where the development efforts could be focused not only on the interest zones of central contributors, shifting priorities in order to build a more socially visible development policy.

Table IV
LAYER STARS SUMMARY STATISTICS

	Bug Stars (AVG)	Bug Stars (STD)
Framework	43.22	380.40
Libraries	68.68	217.68
Runtime	9.44	34.48
Kernel	126.98	92.80

VI. CONCLUSIONS

Our study exposed that the bug lifetime is similar across Android architectural layers. However the framework and Android layer have long lasting bugs. Additionally, our results exposed that even though central developers pay attention to the framework layer of the architecture, the community also looks to the kernel components. We have exposed an apparent misaligned bug closing proportion

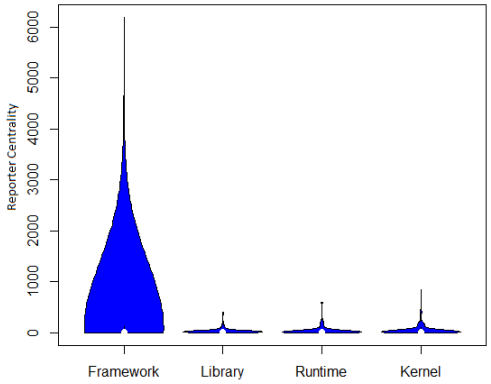

Figure 2. Bug Reporter Centrality (Density) - Android Layer Comparison

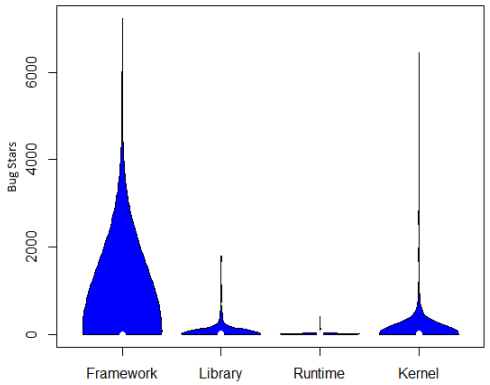

Figure 3. Bug Staring (Density) - Android Layer Comparison

between the social-central and general actors interest zones, and other architectural layers. We believe that strategies like the presented in this paper could be used in order to develop social-centric bug triage and development policies. Our future work pursues cross-referencing current findings with code repositories to include an specific Android developer dimension to our social-centric study.

REFERENCES

[1] J. Śliwerski, T. Zimmermann, and A. Zeller, "Hatari: raising risk awareness," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 107–110.

[2] A. Kumar Maji, K. Hao, S. Sultana, and S. Bagchi, "Characterizing failures in mobile oses: A case study with android and symbian," in *Software Reliability Engineering (ISSRE), 21st International Symposium on*. IEEE, 2010, pp. 249–258.

[3] E. Shihab, Y. Kamei, and P. Bhattacharya, "Mining challenge 2012: The android platform," in *The 9th Working Conference on Mining Software Repositories*, 2012, p. to appear.

[4] J. Hintze and R. Nelson, "Violin plots: a box plot-density trace synergism," *American Statistician*, vol. 52, no. 2, pp. 181–184, 1998.