Characterizing Failures in Mobile OSes: A Case Study with Android and Symbian

Amiya Kumar Maji, Kangli Hao, Salmin Sultana, and Saurabh Bagchi
School of Electrical and Computer Engineering
Purdue University, West Lafayette, Indiana, USA
{*amaji, khao, ssultana, sbagchi*}*@purdue.edu*

***Abstract*—As smart phones grow in popularity, manufacturers are in a race to pack an increasingly rich set of features into these tiny devices. This brings additional complexity in the system software that has to fit within the constraints of the devices (chiefly memory, stable storage, and power consumption) and hence, new bugs are revealed. How this evolution of smartphones impacts their reliability is a question that has been largely unexplored till now. With the release of open source OSes for hand-held devices, such as, Android (open sourced in October 2008) and Symbian (open sourced in February 2010), we are now in a position to explore the above question. In this paper, we analyze the reported cases of failures of Android and Symbian based on bug reports posted by third-party developers and end users and documentation of bug fixes from Android developers. First, based on 628 developer reports, our study looks into the manifestation of failures in different modules of Android and their characteristics, such as, their transience in time. Next, we analyze similar properties of Symbian bugs based on 153 failure reports. Our study indicates that Development tools, Web browsers, and Multimedia applications are most error-prone in both these systems. We further analyze 233 bug fixes for Android and categorized the different types of code modifications required for the fixes. The analysis shows that 78% of errors required minor code changes, with the largest share of these coming from modifications to attribute values and conditions. Our final analysis focuses on the relation between customizability, code complexity, and reliability in Android and Symbian. We find that despite high cyclomatic complexity, the bug densities in Android and Symbian are surprisingly low. However, the support for customizability does impact the reliability of mobile OSes and there are cautionary tales for their further development.**

I. Introduction

In an interview with BBC in February 2008, Andy Rubin, Google's director of mobile platforms, commented "*There should be nothing that users can access on their desktop that they can't access on their cell phone.*" [1]. This vision of bridging the gap between desktop computers and hand-held devices, brings with it new challenges in the expanding smartphone market. As dreams turn into reality, the new age mobile devices have transformed into miniaturized entertainment consoles connected to the global information backbone, the Internet. However, every new feature comes with a cost—cost of new software, cost of memory, cost of battery consumption, user frustration due to failures, etc. In this paper, we analyze the "cost" of smartphones from the point of view of failures. We seek to answer the question how is the reliability of a mobile operating system impacted when the developers offer a feature-rich and highly configurable system, such as Android (from Google and Open Handset Alliance) and Symbian (from Nokia and Symbian Foundation). The relation between complexity and reliability in traditional operating systems has been well studied [2], [3], [4], [5], [6]. But how the evolution of smartphones, with the increase in complexity, impacts their reliability is a question that has been largely unexplored till now. This kind of system introduces an important additional dimension to the question, namely, the significant resource constraints—computation power, memory, battery, and display real estate. Their interface with an wide range of sensors, e.g. camera, accelerometer, and GPS, some of which may bring in copious streams of data, puts additional demands on the mobile OS. With the release of Android, an open-source operating system for mobile devices (open sourced in October 2008) and Symbian (open sourced in February 2010), we are now in a position to analyze their dependability and see what lessons we can draw for modern, highly customizable, mobile operating systems.

Our case study focuses on Android and Symbian in part due to the enormous interest shown by cellphone manufacturers and application developers in them. Symbian at present holds the largest market share for mobile OSes on smartphones (46.9%) [7], whereas Android is predicted to be the second largest mobile OS by 2012 [8]. The programmability features of both these platforms empower us with a high degree of flexibility unforeseen in hand-held devices. We believe that reliability analysis of such platforms will be of value to the developers and researchers in directing their efforts at building a user-friendly *and* robust framework under tight resource constraints.

Several researchers have analyzed failure characteristics of popular operating systems like Windows and Linux [4], [5]. But evaluations of mobile operating systems are rarely seen. A notable exception is the study of failures in Symbian OS-based smart phones [9]. However, since Symbian was not open source at the time of the analysis, the authors of this earlier study were limited in what they could do. They used failure logs from 25 smart phones being used by volunteers participating in the study. Our work is the first of its kind in that it looks into *open-source* mobile OSes and classifies their failures based on a much larger number and greater diversity of failure reports. However, our findings on the recovery actions found by users to be most useful are consistent with those from this earlier study.

In this paper, we analyze the reported cases of failures[1] of Android and Symbian based on the bug reports posted by developers in public forums [10], [11], [12]. Our study covers the period from when the OS was made open source till the present (from October 2008 for Android and from February 2010 for Symbian). Our study looks into the manifestation of bugs in different software modules of these two platforms and identifies which modules are reported to cause the greatest unreliability. Our study indicates that Development Tools, Web Browsers, and Multimedia applications are most error-prone in both these systems. We also present the temporal pattern of bug discovery in Android and find that new releases introduce significant new bugs together with fixing some existing bugs. A study of the G1 user forum for the T-Mobile phone running Android [11] gives us insight about user-visible symptoms of the bugs and their possible recovery actions.

To understand the root causes of some of the bugs, we examine the code modifications needed to fix a subset of the bugs, studying a total of 233 bug fixes for Android. A large fraction (78%) of the bug fixes needed only small changes in code, as opposed to significant code modification, even though the bug reports often indicated significant modification would be needed. By measuring the number of environment variables in the Android platform across its different versions and comparing it with Linux Kernel, we analyze the customizability of Android. This indicates Android's dependence on a few critical variables, which implies that these environment variables need to be set with care; else, significant error propagation can occur. We further measured the code complexity of Android and Symbian by calculating McCabe Cyclomatic Complexity of some projects within them. Despite a high value for the maximum CC, its bug density is lower by an order of magnitude than what would be indicated by traditional software reliability estimates.

Our overall message from the study is promising. Both Android and Symbian have started a creditable approach of open source mobile OS and provide a radically high level of customizability both in building and executing the OS. While this customizability does lead to some loss of reliability, especially in Development tools and Third-party applications, such loss can be mitigated by more rigorous testing. The kernels of both these platforms are significantly hardened (only few bugs per million lines of code), however, we need more efforts in improving the middleware. We believe, if we are to trust the smartphones of today and tomorrow, openness of their development and testing together with scientific analysis of their failures are critically needed.

[1]In this paper, we use the terms *error* and *failure* interchangeably since we consider that all the errors lead to *visible* failures, either at the developer or at the end-user level.

The rest of the paper is organized as follows. We present an overview of the major components of Android and Symbian in Section II. Section III illustrates our analysis of manifestation of bugs in these platforms. In Section IV, we discuss the results obtained from root cause analysis of the bugs, analysis of environment variables, and cyclomatic complexity measures. In the following section, we present a brief summary of the papers that are relevant to our work and compare our results. Finally, we conclude the paper by highlighting the key observations and specifying future research directions.

II. Overview of Android and Symbian

A. Android

The Android platform is a software stack for mobile devices that consists of an operating system, middleware and key applications [13]. Android offers many features covering the areas of application development, Internet, media, and connectivity. These features include Application framework, Dalvik virtual machine, Integrated browser, Optimized graphics, SQLite for structured data storage, Media support for common audio, video, and still image formats, GSM Telephony, Bluetooth, EDGE, 3G, and WiFi, Camera, GPS, Compass, and a rich Development environment. The Android platform primarily consists of five layers:

Applications: This includes a set of core applications that come with the Android distribution like Email Client, Messaging application, Contacts application, Calendar, Map browser, Web browser etc.

Application Framework: This layer has been designed to facilitate the reuse of components in Android. With the help of Application Framework elements (such as, Intents, Content Providers, Views, and Managers) in Android, developers can build their applications to execute on Android Kernel and inter-operate among themselves and with existing applications.

Libraries: Libraries include System C library, Surface Manager, 2D and 3D graphics engine, Media Codecs, the SQL database Sqlite and the web browser engine LibWebCore.

Android Runtime: The Android runtime consists of two components. First, a set of Core libraries which provides most of the functionality available in Java. Second, the Dalvik virtual machine which operates like a translator between the application side and the operating system. Every Android application runs in its own process, with its own instance of the Dalvik virtual machine.

Linux Kernel: Android uses a modified version of Linux 2.6 for core system services such as Memory Management, Process Management, Network Stack, Driver Model and Security. For more information on the Android platform and a schematic of the Android architecture the readers are referred to [13].

B. Symbian

Symbian OS, currently being used by several leading mobile phone manufacturers, account for 46.9% of global smart phone sales, making it the world's most popular mobile operating system [7]. It is a lightweight operating system designed for mobile devices and smart phones, with associated libraries, user interface, frameworks and reference implementations of common tools, originally developed by Symbian Ltd [14].

Since mobile phones' resources and processing environments are highly constrained, Symbian was created with 3 design principles: (i) Real time processing, (ii) Resource limitation, and (iii) Integrity and security of user data. To best follow these principles, Symbian uses a hard real-time, multithreaded microkernel, and has a request-and-callback approach to services. Symbian's system model is segmented into 3 main layers [15]:

OS Layer: Includes the hardware adaptation layer (HAL) that abstracts all higher layers from actual hardware and the Kernel including physical and logical device drivers. It also provides programmable interface for hardware and OS through frameworks, libraries and utilities etc. and higher-level OS services for communications, networking, graphics, multimedia and so on.

Middleware Layer: Provides services (independent of hardware, applications or user interface) to applications and other higher-level programs. Services can be specific application technology such as messaging and multimedia, or generic to the device such as web services, security, device management, IP services and so on.

Application Layer: Contains all the Symbian provided applications, such as multimedia applications, telephony and IP applications etc.

Symbian is optimized for low-power battery-based devices and ROM-based systems. Here, all programming is event-based, and the CPU is switched into a low power mode when applications are not directly dealing with an event. Similarly, the Symbian approach to threads and processes is driven by reducing memory and power overheads. Readers are referred to [15] for further details on the Symbian architecture.

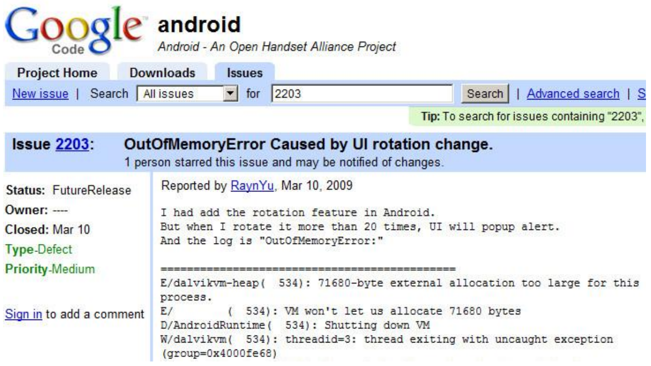
III. Analysis of Bugs in Android and Symbian

Now we characterize the bugs manifested as failures in Android and Symbian.

A. Data Collection

The Android issue reporting site [10] currently contains more than 8300 bug reports that are categorized into two types, namely *defects* and *enhancements*. We consider these as the manifestations of faults from an application developer's perspective. Our study considers only bugs marked as *defects*. Since the "issue" descriptions are stored as unstructured texts and contained varying amount of details, we had to go through them manually. To prune the database further, we used a set of keywords to list bugs containing those tags. These **keywords** are—*crash, shutdown, freeze, broken, failure, error, exception*, and *security*. The motivation behind choosing these keywords is that these events typically represent significant user inconvenience [9]. The dataset was also filtered to remove duplicate entries. This initially gave rise to a list with 758 bugs reported between Nov 2007 and Oct 2009. Since the bugs between Nov 2007 and Oct 2008 are reported before the official release of Android we further removed these pre-release bugs. The final dataset for Android thus consisted of 628 distinct bugs.

To get an understanding of the terminology used in the issue database, let us consider the bug displayed in Fig. 1. This is the entry of a bug related to memory exhaustion (Issue ID 2203) in the Android issue reporting site [10]. The developer claims that when she tries to rotate the UI of Android more than 20 times, it terminates with an "OutOfMemoryAlert". The bug type is specified as "Defect" and it has medium priority. The "Closed" date in the report indicates that the bug has been fixed and it will be released in future (hence, labeled as "FutureRelease").



Tip: To search for issues containing "2203",

Issue 2203: OutOfMemoryError Caused by UI rotation change.
1 person starred this issue and may be notified of changes.

Status: FutureRelease
Owner: ----
Closed: Mar 10
Type-Defect
Priority-Medium

Sign in to add a comment

Reported by RaynYu, Mar 10, 2009

```
I had add the rotation feature in Android.
But when I rotate it more than 20 times, UI will popup alert.
And the log is "OutOfMemoryError:"

=============================================
E/dalvikvm-heap(  534): 71680-byte external allocation too large for this
process.
E/        (  534): VM won't let us allocate 71680 bytes
D/AndroidRuntime(  534): Shutting down VM
W/dalvikvm(  534): threadid=3: thread exiting with uncaught exception
(group=0x4000fe68)
```

Figure 1. A Sample Bug Report on the Android Issue Reporting Site

The different categories of the pruned set of bugs (after further removing unhelpful categories "Questions" and "Declined") with the associated counts are shown in Table I.

We applied the same methodolgy for collecting bugs from the Symbian bug tracker [12] which currently has more than 2700 bugs. Initially the database was queried with the keywords mentioned earlier. The collected failure reports were pruned by removing duplicates and entries of type "enhancement" or "feature". This gave us a dataset with size 275 spanning the period May 2009 — April 2010. After removing the pre-release bugs (before 4 Feb, 2010) our dataset contained 153 distinct failures. The bugs in our Symbian dataset have the status of *new* (the bug has just been reported), *assigned* (the bug has been assigned to an engineer for fixing), *proposed* (a solution has been proposed that is awaiting verification), *closed/worksForMe* (the bug is closed since it could not be reproduced), *resolved/fixed* (the bug has been fixed and the suggested resolution is awaiting verification from the package owner), or *closed/fixed* (the bug had been fixed and the fix is in a release or pre-release version). Table I shows a breakup of the bugs considered in our analysis for each of the two platforms.

Table I
BREAKUP OF BUGS CONSIDERED IN OUR ANALYSIS

Android		Symbian	
New	330	New	106
Assigned	13	Assigned	15
Reviewed	67	Proposed	21
NeedsInfo	9	Resolved/Fixed	5
FutureRelease	119	Closed/WorksForMe	2
Released	69	Closed/Fixed	4
Unassigned	2		
Unreproducible	19		
Total	628	Total	153

Next, we analyze the failure reports from two viewpoints—the first one is to identify the frequency of failures in different segments of Android and Symbian, whereas, the second one is to identify the temporal pattern in bug discovery. We further classify whether the bugs are permanent, intermittent, or transient.

B. Location of Manifestation of Errors

From the details of the bugs, we initially identified the location where a bug is manifested. Notice that 'location' has different interpretations from different perspectives. From a user's point of view, the location of a bug is the application which fails to run correctly, whereas, from a system developer's point of view, the location is the exact component of Android that fails (often found from stack trace). Our analysis presents categorization of bugs primarily from application developers' perspective, however, a small fraction of the bugs are also reported by end-users (containing fair amount of details).

From the bug reports, we first identified 55 different *segments* in Android where bugs were reported. By segment, we mean an individual application or an individual library at which the bug manifested itself, from an end-user's perspective. However, this proved to be too many segments for getting an understanding of the underlying bugs and some segments had very few bug reports. Therefore, we performed an aggregation of some of the related applications and libraries into aggregated segments. Through this we arrived at 18 Android segments. *'Segment'* now represents a built-in application (e.g. Camera, Web Browser etc.), a library in Android (e.g. Graphics, SSL etc.), or an aggregate. The aggregates are: Eclipse, Android Development Tool (ADT), Android Debug Bridge (ADB) as Development Tools; GPS and Location Manager as Location Manager; all the applications that come with Android and are related to

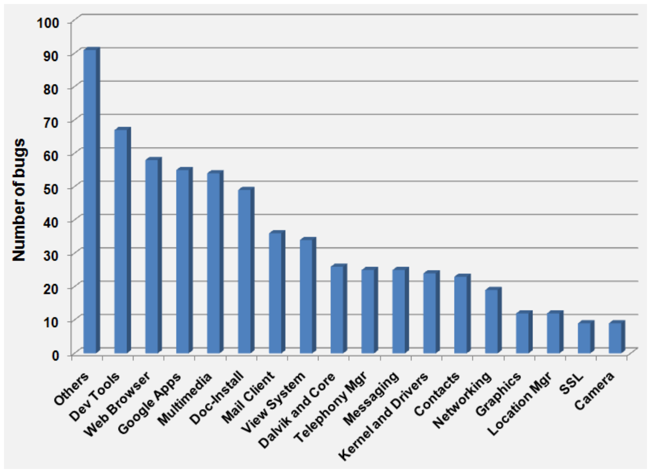


Figure 2. Manifestation of Bugs in Different Segments of Android. The total number of bugs considered here is 628.

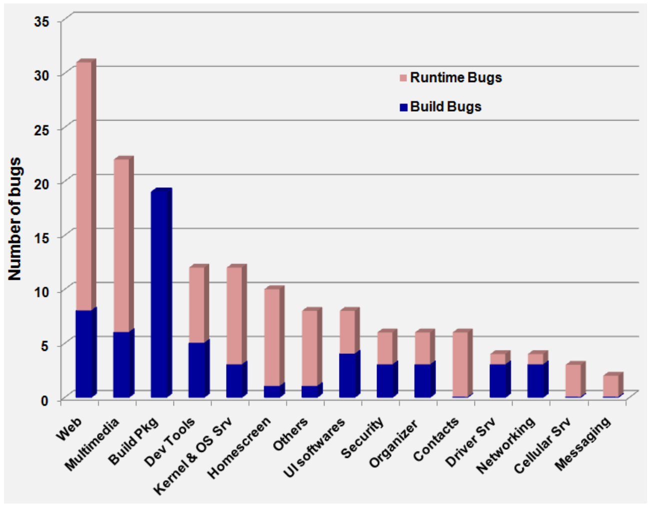


Figure 3. Manifestation of Bugs in Different Segments of Symbian. The total number of bugs considered here is 153.

Google's services, such as Gmail, Map, and Market Place as Google Apps; Networking library and Wi-Fi library as Networking; Image viewer, Media library, and Media player as Multimedia. The obtained results for Android is displayed in Fig. 2. The Y-axis in the graph indicates the count of distinct bugs that were reported against a segment.

A large collection of applications that did not have enough severity individually (examples contain failures of Activity Manager, Content Provider, Memory Manager, SQLite etc.) were merged to form the largest segment "Others" (91) in Fig. 2. The next most failure-prone segment in Android was Development tools (67) followed by other significant segments such as Web Browser (58), Google Apps (55), Multimedia (54), Documentation and Installation related bugs (49), Mail Client (36), and the View System (34). It is encouraging to find that a relatively few numbers of bugs are related to Kernel and Device drivers (24 of 628), and the Dalvik and Core Library (26 of 628).

A similar analysis of the Symbian bugs initially resulted in a distribution of 153 bugs in 41 segments ("packages" in Symbian terminology). To get a better understanding we again combined the related packages into a single segment, e.g., the packages *wrttools, web, websrv,* and *webuis* as Web; the packages *podcatcher, mm, graphics, imagingext, mmappfw,* and *musicplayer* as Multimedia; the packages *homescreen* and *homescreensrv* as HomeScreen. This resulted in 15 segments of which only 3 are individual packages (these are messaging, contacts, and organizer) and the rest represent groups of related packages.

It can be seen from Fig. 3 that the segment Web (31) is most bug prone in Symbian followed by Multimedia (22). Bugs related to building of Symbian packages (19), bugs in the Development tools (12), and bugs in Kernel and OS Services (12) are also significant in number. During our analysis, we observed another interesting pattern in Symbian—as many as 59 bugs in our data set (38.6% of all the bugs) were due to build/compilation errors, missing files, missing references, etc. We denote these types of bugs as "build" bugs. Some of the packages contained significant numbers of "build" bugs. Examples include web (8 of 31), Multimedia (6 of 22), Development tools (5 of 12), and UI softwares (4 of 8). We display the relative counts of "build" bugs and "runtime" bugs by splitting the bars in Fig. 3. Note that the "Build Pkg" in Fig. 3 represents bugs specific to the named package and it forms only a fraction of the "build" bugs. The number of "build" bugs in Android did not have such prominence, hence, we do not display the breakup in our analysis. We attribute the large number of build bugs in Symbian to its recent release and believe that these will have less prominence in future releases.

By comparing Fig. 2 and Fig. 3 it can be seen that of the top 6 bug prone segments in both the platforms, 4 are identical. These are Web browser (*Web* in Symbian), Multimedia, Development tools and Doc-Install (*Build* in Symbian). Interestingly, web browsing and multimedia are perhaps the most significant features of a smartphone as perceived by the users. Unfortunately, these are also most failure prone leading to dissatisfaction of users as seen in numerous posts in the user forums. We further note that the Web browsers in both Android and Symbian are built from the WebKit engine. This raises concern about the reliability of third-party applications that are used in the mobile OSes. Apart from WebKit, SQLite, SSL, and Graphics (based on OpenGL) were also found to be error prone in Android.

The presence of large number of bugs in Development tools in both these platforms (10.67% and 7.84% of all bugs in Android and Symbian respectively) draws special attention to this segment. We believe, the efficiency and reliability of these development environments will be a key factor in determining which platform has a larger developer community. Moreover, faults in the development environment can significantly affect the performance of applications and may even be responsible for creating security holes (e.g., a development tool that does not check for known vulnerabilities like SQL Injection and Cross-site scripting). An encouraging finding from our analysis is that both the platforms have lesser number of errors in the lower layers—*Dalvik and Core*, *Kernel and Drivers* in Fig. 2 and *Kernel and OS service*, *Driver services* in Fig. 3—compared to application level failures. Android, which comes with more applications than Symbian, also has more application level failures (like Mail client and Google apps in Fig. 2).

C. Temporal Analysis of Bugs

In this section, we present two types of statistics about failures in mobile OSes. The first one looks at persistence of the bugs – i.e. whether these are environment-dependent or not and transient or non-transient. The other statistic finds out the pattern of bug discovery and fixes with different releases.

1) Persistence of Bugs: From the failure reports, we found that only a few bugs in Android were transient (10) or intermittent (49). Most of the failures are permanent (566) in nature and need to be fixed by modifying the Android code. A few bugs (3) could not be categorized due to lack of sufficient information. It may be noted here that the actual number of transient and intermittent bugs may be much higher as users often refrain from reporting them. Furthermore, many non-permanent bugs were also declined by Android engineers as they could not be reproduced. Similar analysis with the Symbian bugs indicated that only 4 were intermittent bugs and the rest (149) were permanent. We observe that the large number of permanent bugs in both systems (90.12% in Android and 97.38% in Symbian) may be due to the fact these are new operating systems and their codebases are not yet stable.

2) Discovery of Bugs over Time: We counted the numbers of bugs reported every month to find the temporal pattern in bug discovery. The result of the analysis for Android is shown in Fig. 4. Since Symbian has been open-sourced recently (3 months ago), there is insufficient data for temporal analysis. The labels 1–6 in the figure indicate various releases of Android. All the pre-release bugs have been combined into a single column. It can be observed that the peak occurs in Oct 2009, one month after the release of Android 1.6. (Since, Android 2.0 was released on October 26, 2009, we do not consider bugs in October 2009 to be a consequence of this release.) It can also be observed that the Dec 2008 surge follows the release of Android 1.0 R2 and May 2009 surges follows the release of Android 1.5. This points to the fact that each release fixes some bugs and creates new bugs. We observed similar pattern in G1 user forums where many customers complained that they faced new failures after updating their firmware. The heights of the bars are non-monotonic over time, whereas in the ideal case they should be decreasing. This also points to the need for a feature to roll back to a previous version when users face problems on their specific smartphone after upgrading. Due to the lack of version information in most bug reports, we cannot quantify how many bugs were fixed and how many new bugs were generated through the release of an Android version.

For Symbian, the counts of bugs in the months following its release are: 42 in Feb 2010, 63 in Mar 2010, and 48 in April 2010. Though this data is insufficient to draw any significant conclusions, the initial release of Android also showed similar pattern in bug discovery (27 in Nov 2008, 41 in Dec 2008, and 21 in Jan 2009).

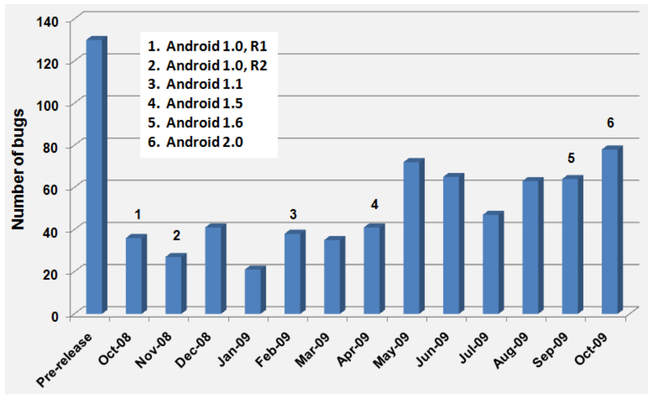


Figure 4. Discovery of Bugs over Time for Android

D. Analysis of User Forums

Besides the developers reports, we also studied publicly available data on the T-Mobile G1 user forums for incidence of Android failures [11]. Our analysis considered threads related to *Messaging, Google Applications, Phone & Data Connection,* and *Operating System & Software Development*. It was observed that most of the reports in the user forums are trivial questions or suggestions for enhancements. Therefore we discarded these messages from our dataset. The final list consisted of 105 distinct failures. The failures are frequently reported for Mail Client (15), SD Card (11), Media Player (9), Messaging (9), GPS and Location Manager (8), Web browser (8), Android Marketplace (6), and Calendar (5). This result is not identical to that when we considered the failure reports from a wider audience (Section III-B). This may indicate that the scope of problems in the different modules varies with the hardware device being used. We further noted that the common user-initiated recovery actions are—*Restart application, Wait for some time, Restart phone, Modify settings, Factory reset, Take out battery, Update firmware,* and *Use third-party software*. For example, many users reported that the location displayed in the GPS of Android is 1-2 miles away from where they actually were. Waiting for sometime, rebooting the phone, or doing factory reset may solve the problem, but they do not work on every G1 phone. These findings about user-initiated recovery actions are consistent with those in the prior work on the Symbian OS [9] in terms of the categories.

IV. Analysis of Code Modifications and Fixes

The statistics presented in the previous section primarily considered manifestation of failures. Though valuable for identifying the impact of bugs in various segments of mobile OSes, it does not give any indication about how these bugs originated. It is therefore necessary to study the root causes of the bugs and to correlate the user-visible failures with these root causes. This may also help us in identifying error propagation. With these objectives, we studied the code modifications in some of the bug fixes and gained useful insights about the failures. Of the two mobile OSes analyzed in this paper, we have details of bug fixes only for Android [16].

A. Data Collection

For this work, we looked into the Android code repositories to study the bug fixes. The code reviews stored in [16] presented us with the details of the fixed bugs. It is to be noted that the failures analyzed in this section have some overlap with the failures studied in section III but do not form a strict subset. Several bug-fixes were found which did not appear in the Issue listing site and vice-versa [10]. Our dataset for this analysis contains 233 bug-fixes from 29 projects in the Android repository. These bugs were fixed during the period October 2008 to October 2009. The significant projects within this collection, in terms of the number of bugs, are `kernel/common`, `kernel/msm`, `kernel/omap`, `platform/framework`, `platform/dalvik`, `platform/build`, `platform/system`, and some applications in `platform/packages`. An exhaustive listing of all the Android projects may be found in [17]. In the following sub-section, we present our analysis of root causes of these bugs.

B. Categorization of Code Modifications

It was observed from the bug-fixes that most of the bugs (179 of 233) required only few lines of code changes. Among the minor modifications, we further identified what types of code changes were most frequent. We categorize different types of code modifications as follows.

1. *Major:* Fixes that involve modifications of more than 10 lines of code, or modifications at more than 5 places in the source file(s).
2. *Add/modify attr val:* Update the value assigned to a variable (e.g. the code `A.x=B.y` is corrected as `A.x=C.z`) or declare a new variable.

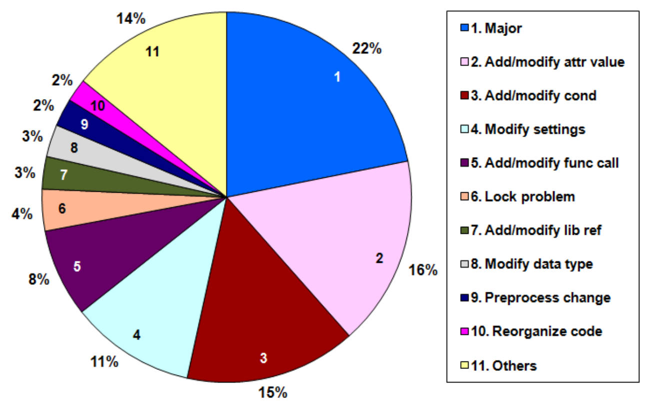


Figure 5. Different Types of Code Modifications. Total number of modifications considered for this analysis was 233.

3. *Add/modify cond:* Add some new checks (if-stmt), or add a missing else clause, or modify the condition expression.
4. *Modify settings:* Update system constants or include modification in the makefiles, application configuration files etc.
5. *Add/modify func call:* Introduce a new function call, or modify the arguments of an existing invocation.
6. *Lock problems:* Bug was caused because a critical segment was not locked or a lock was not removed and deleted upon exit.
7. *Add/modify lib ref:* Bug was caused because code was accessing some non-existent or incorrect libraries or classes.
8. *Modify data type:* Update the datatype of a variable.
9. *Preprocess change:* Introduce a preprocessor directive (e.g. adding an `ifdef`).
10. *Reorganize code:* Change the order of execution of certain code blocks.
11. *Others:* The bug fixes that could not be placed under any of the above-mentioned categories are considered here.

In our analysis, some fixes contained multiple changes (e.g. some bugs needed both modification of settings and addition of new attribute values), hence, they were considered under both the categories. We show the breakup of the different bug fix categories in Fig. 5. We observe that only 22% of the fixes required major changes. These modifications primarily include addition of new functions, data structures, and constants. Among the rest, most of the fixes were of the type *Add/modify attr value* (16%) and *Add/modify cond* (15%). The large percentage of *Add/modify attr val* (16%) arises due to the fact that some applications/drivers in Android are still undergoing major code revisions. The list of changes may be seen from the release notes of different versions of Android. Within the sizable category *Add/modify cond*, we observed several instances where an *if-stmt* did not have a corresponding *else* clause. This resulted in exceptional cases not being handled correctly. Detailed specification of the program behavior could have avoided such errors. It is known that introducing new conditional statements adds to the cyclomatic complexity of a program. Hence, while implementing these fixes, the designers must be careful so that understandability and testability of the resulting code is not altered significantly, e.g., if a fix introduces a high degree of nesting for conditional statements, the designers may try to simplify by reorganizing the code.

C. *Tension between Customizability and Reliability*

The presence of 11% *Modify settings* bugs motivated us to delve deeper into the Android code base and analyze the flexibilty provided by Android runtime environment to the upper layer applications. The customizability claim is buttressed by the fact that Android may be adapted for a wide range of mobile hardware (people have even used it to run notebooks), it incorporates virtual machine, Sqlite for data storage, and the fact that people may use their phone to write programs.

We observed that a large chunk of the failures requiring *Modify Settings* surfaced during building (compiling) the Android source code. This resulted in modifications of the Makefiles to support new architectures and APIs, modification of environment variables, changing application permissions etc. Note that many of the application configuration files are also generated during the build process itself. Hence, we consider that *Modify settings* errors relate to customizability of the system, where customizability is defined to include both the process of building the system and of executing the system. However, this category of bugs only covers a subset of the errors caused by the need to support customizability. According to our categorization, if a new condition needs to be introduced to handle a configuration parameter, that would be classified under *Add/Modify cond*, while some support for customizability may necessitate large changes in the software (considered under the category *Major*). Thus, the percentage of bugs to support customizability is non-trivial. This suggests that customizability does have some negative impact on the reliability. But this is not egregiously high for the level of customizability supported by Android. Further, improvements in software practices, especially targeted at the problematic segments that we have identified here, plus a natural maturing of the code base (recollect that we are talking of something that has been open sourced for only about a year and a half) will likely bring these bug incidences down.

D. *Analysis of Environment Variables*

We observe that applications often read system configurations and locations of various executables from the environment variables defined in the runtime kernel. These parameters, though not comprehensive, are a significant indicator of a framework's customizability. Extending from the conclusions in the previous section, we counted the number of environment variables defined in the Android

Table II
COUNT OF ENVIRONMENT VARIABLES AND THEIR REFERENCES IN DIFFERENT VERSIONS OF ANDROID AND THE LINUX KERNEL

	# env vars	Total refs	Max ref
Android 1.1	62	819	577
Android 1.5	63	854	584
Android 1.6	76	1545	584
Android 2.0	82	2083	592
Linux Kernel 2.6.32	127	953	158

platform and compared it with a standard Linux Kernel (version 2.6.32). We wrote a script to scan the source codes of different versions of Android to find the occurrence of `export` or `setenv` keywords. We then built lists of environment variables for each of the Android versions and the Linux Kernel. Next, we counted the references to each environment variable and summed them up. Though a variable may be referred multiple times within a single line, we consider these as a single reference. The results obtained from our analysis are presented in Table II. "Max ref" is the maximum number of references to a single environment variable in the entire code base.

The number of environment variables in different versions of Android is steadily increasing. Though this number (82 in Android 2.0) is lower than in the Linux Kernel (127), it is still significant considering that Android is built as a mobile OS and runs on devices with more constrained resources than the Linux kernel. Also, the growth in the number of references to environment variables between February 09 (Android v. 1.1) and October 09 (Android v. 2.0), 154%, is striking evidence of the rapid march toward a customizable mobile OS. More than 85% of the references to all the environment variables were made from codes in `external/` folder in Android which includes third-party libraries and built-in applications.

In Fig. 6, we illustrate the distribution of the number of references to environment variables. The number of variables with reference count of zero indicates that a large number of environment variables were not referenced outside the line where they were defined or "exported". We also noted that majority of the references were made to only a few of the variables (less than 5). For example, in Android 1.6, the environment variables `DESTDIR`, `MK`, `CFLAGS`, and `LDFLAGS` are referenced 584, 308, 194, 117 times respectively. The maximum reference count for a single environment variable was much higher in Android than in Linux Kernel. The references to environment variables in Linux Kernel are almost evenly distributed, whereas, Android is more dependent on a few critical variables. This indicates in Android a possibility of significant error propagation if these key environment variables happen to be incorrectly set. As a corollary, Android will benefit from building in reasonableness checks for these key environment variables.

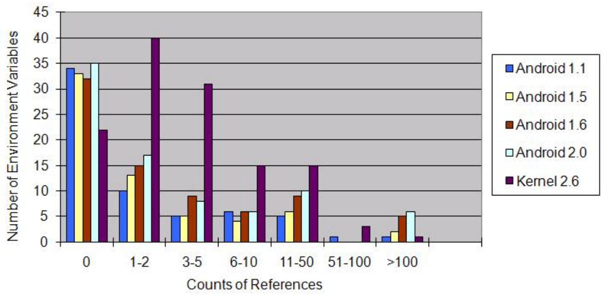


Figure 6. Distribution of References to Environment Variables

E. Cyclomatic Complexity and Number of Bugs

Cyclomatic complexity, which measures the number of linearly independent paths through a program's source code, is frequently used as a metric of code complexity. To understand the relation between cyclomatic complexity and bug density in Android and Symbian, we selected a set of projects (packages) in these two platforms that had the maximum number of bugs. Since, the Android issue reporting site [10] does not contain the root cause of a bug, we considered the bug counts in section IV for computation of bug density. For Symbian, we found that each bug was assigned to its corresponding project. Hence, the bug density in Symbian is computed with the counts presented in Fig. 3. We computed cyclomatic complexities (CC) of these projects (packages) with Understand 2.0—a source code analysis and metrics generation tool [18]. The unit for computation of cyclomatic complexity is a function, as is typically done. Source codes for these projects (packages) were downloaded from Android [17], and Symbian [19] repositories. Note, that the Android repository gives us the latest source codes, whereas, our bug pool is older than the source code. Hence, the calculated bug density is not completely consistent. Nevertheless, this gives an approximate measure of code complexity in Android and may be used to suggest improvements in code quality. Table III and Table IV show the results obtained from our analysis.

It was observed that in Android many projects in `kernel/*` have large overlap in their code files. As a result, the Max cyclomatic complexity and Source Lines of Code (SLOC) of these projects are similar or identical. Qualitatively, the bug density is quite low for both these systems indicating a high standard of code development, even though these are relatively new software projects. For reference, the pre-release bug density in Windows XP was 2.66×10^{-3} [20]. Furthermore, the bug density in the Kernel and OS Services of Symbian was lower compared to Android Kernel.

In standard literature, it is suggested that cyclomatic complexity of functions should be limited to 20 for man-

Table III
CYCLOMATIC COMPLEXITY AND BUG DENSITY OF DIFFERENT PROJECTS IN ANDROID

Projects	**Bug Density** $\times 10^4$	**No. of bugs**	**Source LOC**	**Avg. Cyclomatic**	**Max Cyclomatic**
kernel/omap	0.04	21	5,311,427	1.12	4,973
kernel/msm	0.06	29	4,724,260	5.60	4,973
kernel/common	0.07	31	4,688,175	5.82	4,973
dalvik	0.18	14	771,865	2.23	766
development	0.46	10	216,344	2.18	169
framework/base	0.79	51	645,978	2.40	221
packages/apps/camera	1.33	2	14,962	2.15	20
packages/apps/mms	1.74	4	23,013	2.02	46
system/core	1.90	13	68,798	4.31	167
hardware/msm7k	2.42	3	12,382	4.00	23

Table IV
CYCLOMATIC COMPLEXITY AND BUG DENSITY OF DIFFERENT SEGMENTS IN SYMBIAN

Segments (Fig. 3)	**Bug Density** $\times 10^4$	**No. of bugs**	**Source LOC**	**Avg. Cyclomatic**	**Max Cyclomatic**
Kernel and OS Services	0.03	12	3,684,192	3.02	1,470
Security	0.08	6	752,148	2.29	134
Multimedia	0.12	22	1,866,577	2.44	558
Web	0.17	31	1,807,828	3.01	2,442
HomeScreen	0.38	10	263,305	2.25	149
Build Pkg	0.63	19	299,868	2.24	268

ageability of code. Hence, the Max. cyclomatic complexity figures presented in the table may initially appear erroneous. A careful examination of the source code, however, revealed that some functions extensively use macros inline, each of which contains multiple if-else statements. When the macros are replaced by the preprocessor with their corresponding codes this gives rise to high cyclomatic complexity. Such inlining is the main reason for our high cyclomatic complexity in both these tables. We further noted that large case-switch statements were, in many cases, responsible for cyclomatic complexity above 100. The presence of long case-switch statements necessitates the creation of extensive code documentation explaining each of the cases. Verifying that all the cases have been handled properly is a challenging task. Going forward, as mobile OSes increases in complexity, developers will need to pay careful attention to managing the long switch statements.

Average cyclomatic complexity (CC) per function, on the other hand, was significantly lower (between 1.12 and 5.82 in Android and between 2.24 and 3.02 in Symbian). This is primarily due to the presence of a large number of default and inherited functions which have complexity 1. We also measured the CC of the Linux kernel (version 2.6.32). This system consisting of 6,082,112 lines of source code had a maximum CC of 4,973 which is identical to the Android kernel. This can be explained by the fact that Android kernel is built using a modified version of Linux kernel (v2.6). It was observed that the max CC in Symbian Kernel and OS Services is much lower than that in Android and the Linux Kernel, while their average CCs are comparable.

V. RELATED WORK

The goal of research in software reliability analysis has been to classify software errors, as well as to characterize various properties of failures. Such characterization enables us to not only assess the effects of failures but also to prevent and detect new bugs. Reliability literature over the years contains the results of many research efforts directed at analyzing bug reports for popular operating systems [2], [4], [5], [6]. In one of the early works on OS reliability, Sullivan *e*t al. [2] analyzed defects of the MVS operating system based on empirical failure records documented by field IBM staff. This work categorized defects as overlay (errors that corrupt memory) and regular (those that do not corrupt memory). The frequency and effects of both types of errors were analyzed.

Chou *e*t al. , in [4], presented their finding on OS errors by compiler attachments, which checks code for certain types of bugs, and counts bug density. The work discovered the correlation of different types of bugs with directories, function size, and file age. Our work looks into similar problem with different scope: instead of compiler attachments, our research is based on the developers' view and how bugs are fixed. In [6], the authors highlight the failure characteristics of BlueGene/L supercomputer by correlating data obtained from event loggers. Applying similar concept of failure event loggers, Cinque *e*t al. [9] performed one of the few pieces of work that focuses on mobile OS reliability. In this work, the authors attached fault event loggers to a set of 25 Symbian OS based mobile phones to record failure events and panics (kernel-generated warnings for Symbian). Through this, the authors unveiled characteristics of the panics (burst, etc.)

and the relation between panics and user-visible failures. However, since Symbian was not open-sourced at the time of this analysis, their research was limited to the manifestation of failures. Our paper extends the scope beyond these findings by classifying failures according to their root causes in the source code (for Android). We also present an analysis on customizability and complexity of a mobile OS (Android) which is distinct from previous work.

Our work is also comparable to the failure modes catalog for wireless applications presented by Jha *e*t al. [21]. In this work, the authors created a catalog for risk based testing of wireless applications based on observed and predicted failures. Our work also tries to estimate the likelihood of failure at different segments of a mobile OS which can be used for risk analysis. Comparing to this catalog [21], most of the bugs observed in our analysis were found under the category "*Product Elements*" which deals with mobile middleware, platform, synchronization, memory management etc.

VI. Conclusion and Future Work

Our work is a step toward the failure characterization of an OS for mobile phones. We presented a measurement based failure analysis of two operating systems—Android and Symbian—by studying publicly available bug databases. The key findings are: (1) Most of the bugs (more than 90%) in both these platforms are permanent in nature, suggesting that the codebases are not yet mature. (2) The Kernel layer in both the platforms are sufficiently robust, however, much effort is needed to improve the Middleware (*Application Framework* and *Libraries* in Android). (3) *Development tools*, *Web*, *Multimedia*, and *Build failures* are most prevalent in both the platforms. This suggests the necessity for a better mobile application development tools and need for caution in using third-party libraries. (4) Android offers a great degree of customizability in both the build and the execution processes. This customizability comes at a cost for a significant fraction of bugs—between 11% and 50% (assuming all of *Modify settings*, *Add/modify cond*, *Preprocess changes*, and *Major* changes are due to customizability). At present, the percentage of build errors is also high in Symbian (38.6%). (5) According to our analysis, a significant minority of the bugs in Android (22%) needed major code changes. Among various types of code modifications, fixing variable assignments and control flow update (adding if-else clause) are most widespread.

Our study also highlights the significant contributions of Android and Symbian in the field of mobile OS development. Both these platforms have established well-defined and well-maintained open mechanisms for reporting and dealing with bugs, without which a study like ours would not have been possible. This level of transparency is a pioneering and creditable effort in the smartphone world.

Our future work will focus on analyzing the propagation of errors among various layers, more specifically, between the Middleware and the applications. We also want to more fully explore the relationship between customizability, complexity, and reliability in a mobile operating system.

References

[1] Google bets on Android future. `http://news.bbc.co.uk/2/hi/technology/7266201.stm`

[2] M.Sullivan and R.Chillarege, "Software Defects and their Impact on System Availability - A Study of Field Failures in Operating Systems," In Proc. of 21st International Symposium on Fault- Tolerant Computing (FTCS), P. 2–9, 1991.

[3] S.Chandra and P.M.Chen. "Whither Generic Recovery from Application Faults? A Fault Study using Open-Source Software," In Proc. of the 30th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2000, P. 97–106, 25-28 June 2000.

[4] A.Chou, J.Yang, B.Chelf, S.Hallem, and D.Engler. "An Empirical Study of Operating Systems Errors," In Proc. of 18th ACM Symposium on Operating Systems Principles (SOSP), P. 73–88, 2001

[5] W.Gu, Z.Kalbarczyk, R.K.Iyer, and Z.Yang. "Characterization of Linux Kernel Behavior under Errors," In Proc. of International Conference on Dependable Systems and Networks (DSN), 2003, P. 459–468, 22-25 June 2003.

[6] Y.Liang, Y.Zhang, M.Jetta, A.Sivasubramaniam, and R.Sahoo. "BlueGene/L Failure Analysis and Prediction Models," In Proc. of International Conference on Dependable Systems and Networks (DSN), 2006, P. 425–434, 25-28 June 2006.

[7] Gartner Says Worldwide Mobile Phone Sales to End Users Grew 8 Per Cent in Fourth Quarter 2009; Market Remained Flat in 2009. `http://www.gartner.com/it/page.jsp?id=1306513`

[8] Gartner Inc., "Android to overtake iPhone in 2012," October 2009, `http://www.computerworld.com/s/article/9139026/Android_to_grab_No._2_spot_by_2012_says_Gartner`

[9] M.Cinque, D.Cotroneo, Z.Kalbarczyk, and R.Iyer. "How do Mobile Phones Fail? A Failure Data Analysis of Symbian OS Smart Phones," In Proc. of International Conference on Dependable Systems and Networks (DSN), 2007, P. 585–594, 25-28 June 2007.

[10] Android bug listing. `http://code.google.com/p/android/issues/list`

[11] T-Mobile G1 Forum. `http://forums.t-mobile.com/tmbl/?category.id=Android`

[12] Symbian Bug Tracker. `http://developer.symbian.org/bugs/`

[13] What is Android? `http://developer.android.com/guide/basics/what-is-android.html`

[14] Symbian OS on Wikipedia. `http://en.wikipedia.org/wiki/Symbian_OS`

[15] Symbian System Model. `http://developer.symbian.org/wiki/index.php/Symbian_System_Model`

[16] Android Code Review. `https://review.source.android.com/Gerrit #all,merged,n,z`

[17] Android Source Code Repository. `http://android.git.kernel.org/`

[18] Understand 2.0: A source code analysis tool. `http://www.sci-tools.com/ products/understand/`

[19] Symbian Source Code Repository. `http://developer.symbian.org/main/source/packages/index.php`

[20] O.H.Alhazmi, Y.K.Malaiya, and I.Ray, "Measuring, Analyzing and Predicting Security Vulnerabilities in Software Systems," Computers & Security Journal, Volume 26, Issue 3, May 2007, pp. 219–228.

[21] A.Jha and C.Kaner, "Bugs in the brave new unwired world," Pacific Northwest Software Quality Conference, Portland, OR, October 2003.