Testing for Poor Responsiveness in Android Applications

Shengqian Yang Dacong Yan Atanas Rountev
Department of Computer Science and Engineering
Ohio State University, Columbus, Ohio, USA
{yangs,yan,rountev}@cse.ohio-state.edu

***Abstract*—An important category of defects in Android applications are related to *poor responsiveness*. When the user interface thread performs expensive operations, the application is sluggish and may fail with an "Application Not Responding" error. Poor responsiveness has serious negative consequences for user perception and marketplace success. We propose a systematic technique to uncover and quantify common causes of poor responsiveness in Android software. When test cases are executed against the application GUI, artificial long delays are inserted at typical problematic operations (e.g., at calls that access the network). This *test amplification* approach may exhibit increased response times for GUI events, which demonstrates the effects of expensive operations on poor responsiveness observed by the user. The proposed approach successfully uncovered 61 responsiveness problems in eight open-source Android applications, due to inappropriate usage of resources such as network, flash storage, on-device database, and bitmaps.**

I. Introduction

At present, Android is the leading platform in the smartphone market worldwide [11]. Android devices such as tablets and e-readers (e.g., Google Nexus, Amazon Kindle Fire HD) are also rapidly increasing in popularity. Growing numbers of users are employing Android devices in their everyday lives, for diverse tasks such as access to and sharing of information, online purchases and other monetary transactions, casual gaming, and playback of audio and video. The pervasive use of Android applications, together with their increasing popularity, require comprehensive and sustained efforts to improve software quality. Currently there are few software testing techniques designed for this domain. There is clear need for new research investigations and effective tool support for testing of Android software.

A. Poor Responsiveness due to Jank in Android Applications

A category of defects with highly-undesirable consequences are related to *poor responsiveness*. If an application takes more than 200 ms to respond to a user event, it is perceived to be sluggish and unresponsive [10]. The worst-case scenario is an "Application Not Responding" (ANR) error, displayed by the Android runtime when the application is deemed to have stopped responding (e.g., no response to a key press within 5 seconds) [20]. Poor responsiveness has direct effect on the user's perception of the application—frequent sluggishness and ANR messages would motivate the user to uninstall the application, and possibly submit a low rating and negative comments in the app market [12]. Thus, avoiding and fixing responsiveness defects is a very high priority for developers [10], [20].

The reason for poor responsiveness is well known: the main thread of an Android application is the one that processes UI events, and this thread should *not* perform heavy computations or long-wait operations in response to user events. Typical examples of such heavyweight processing are network operations, database operations, file I/O, and bitmap processing; in the Google developer community, they are colloquially referred to as "jank" [10]. Testing for the presence and effects of jank is challenging. For example, in a development environment where the application runs in an emulator, network access typically is very fast and network-related jank may remain unnoticed, while on the actual device this access could be through a much slower 3G wireless network. Although there are informal guidelines to avoid poor responsiveness, this important problem has not been investigated by the software engineering research community, and currently *there do not exist any testing strategies for exposing jank in Android applications.*

B. Our Proposal

We propose a systematic technique to uncover and quantify suspicious behavior that may lead to poor responsiveness in Android applications. Based on a graph model of the GUI of the application, test cases are executed to cover each GUI state and state transition. These tests are then re-executed in a modified environment in which artificial long delays are inserted at typical sources of jank (e.g., around code that accesses the network). This approach, referred to as *test amplification*, may exhibit increased response times when certain GUI transitions are traversed. Such increases are used to characterize the effects of jank: the relationship between different inserted delays and the resulting GUI response times quantifies the externally-observable effects of "janky" operations. This characterization, together with the test case and the amplified application code, are reported to the tester as evidence of poor responsiveness. This approach demonstrates a direct cause-effect relationship between expensive operations and poor responsiveness observed by the user. In our experiments this technique successfully uncovered 61 problems in eight open-source Android applications, due to inappropriate usage of resources such as network, flash storage, on-device database, and bitmaps.

978-1-4673-6333-4/13/$31.00 © 2013 IEEE

MOBS 2013, San Francisco, CA, USA

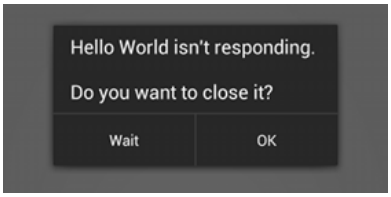
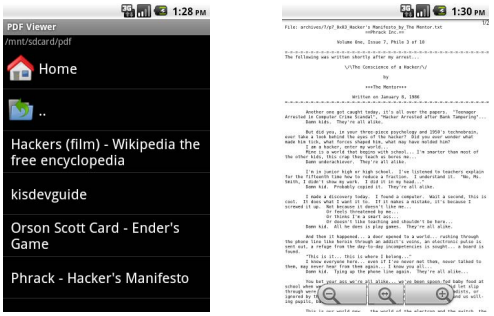


Fig. 1. ANR dialog [20].



(a) (b)

Fig. 2. `APV` application: (a) `ChooseFileActivity` lists files and folders. (b) `OpenFileActivity` displays the selected PDF file.

Contributions. The contributions of this work are:

- *Test amplification criteria:* We define several test amplification criteria for different categories of problematic operations. This approach is based on common causes of jank in Android applications.
- *Test amplification and execution:* We describe how test cases are amplified and executed in order to uncover janky operations and to quantify their effect on responsiveness.
- *Evaluation:* We evaluate the proposed approach on eight open-source Android applications. The evaluation shows that a wide variety of expensive operations are commonly found in the UI thread, and that our amplification approach effectively exposes these janky operations.
- *Case studies:* We describe several case studies of responsiveness defects. These studies, together with the observations from the rest of our experiments, are used to suggest directions for future work on the important problem of poor responsiveness in Android software.

These contributions add to the growing body of work in the emerging area of analysis and testing of Android software. The experimental results and case studies increase our understanding of common defects in mobile applications, and point to interesting problems for future research.

II. Background

A. Poor Responsiveness in Android Applications

An "Application Not Responding" (ANR) error, illustrated in Figure 1, is the most visible manifestation of poor responsiveness. If the application does not respond to user input (e.g., screen touch) within 5 seconds, the Android runtime presents an ANR dialog to the user [20]. Such errors create a highly-negative user experience, and avoiding them is very important [20]. Even less extreme poor responsiveness can be very damaging: a delay of more than 200 ms creates the perception of stuttering, being sluggish, or freezing [10], which may cause the user to uninstall the application and to submit a negative comment in the app marketplace.

The cause of this highly-undesirable behavior is colloquially referred to as "jank" by Google engineers [10]. Jank is excessive work performed in the event-handling thread, which in Android is the main thread of the application. A typical example is a network operation in an event handler. In a wireless 3G environment, a simple "ping" could be as slow as 800 ms, and an HTTP fetch of a small amount of data can take seconds [10]. Another example are I/O operations on the flash file system. The performance characteristics of flash storage differ significantly from those of a traditional disk file system. For example, the cost of a simple disk operation can be substantial, and this cost increases significantly when the storage is nearly-full. Similar considerations apply to SQL operations on the built-in SQLite database. As yet another example, jank could come from computationally-expensive operations related to bitmaps [20].

Testing for jank presents a number of challenges. First, the execution environment during software development may differ significantly from the real-world deployment environments. In development, the application is typically run in the standard Android emulator on a desktop machine, and network operations and file I/O operations may be much faster than on actual devices in the hands of users, over a poor network and with nearly-full flash storage. As a result, jank may remain unnoticed until the application is distributed to users. Second, there is no systematic testing strategy that targets the typical sources of jank, and there are no tools to perform such testing automatically. The significant negative effects of poor responsiveness, and the lack of any comprehensive testing techniques and tools, motivate our work on techniques and tools to uncover and quantify the causes of poor responsiveness.

B. GUI-Based Testing

The approach we propose is based on test cases that are executed (and re-executed) against the GUI of the application under test. An Android *activity* is a GUI window. A GUI event generated by the user (e.g., tapping on a button) can trigger a transition to another activity. For illustration, Figure 2 shows screenshots of two activities from the `APV` [4] PDF viewer application. `ChooseFileActivity` is presented to the user when the application starts. Given the list of files and folders, the user can select a PDF file which is then displayed by `OpenFileActivity`. Each activity represents a different GUI state.

We consider testing based on a GUI model. Such a model is available from design documents or obtained through GUI reverse engineering [23], [14], [1], [3], [35]. A partial GUI model for `APV` is shown in Figure 3. (The figure shows only

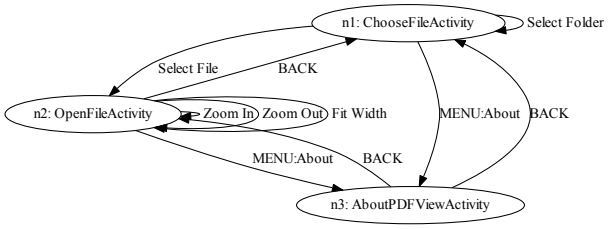


Fig. 3. A subset of the GUI model for APV.

a subset of GUI states and transitions.) The model is a graph where nodes are GUI states and edges are transitions triggered by GUI events. In addition to traditional events, we consider Android-specific events. For example, a user can press the hardware MENU button and select a menu item from a list specific to the current state. In Figure 3 edges labeled with "MENU:" represent such events. As another example, the hardware BACK button can be used to destroy the current activity and to transition to another one.

Various techniques (e.g., [23], [14], [1], [3], [35]) can automatically construct GUI models similar to the one from Figure 3. This process is also referred to as "GUI ripping". An example of a GUI ripping tool for Android is AndroidRipper [1]. In this tool, at each GUI state, the widgets and the events that can be triggered on them are considered at run time. To construct the GUI models used for our test generation, we currently use AndroidRipper; in cases where the tool has deficiencies, manual steps are applied. Clearly, a comprehensive model would lead to more thorough testing. Some very recent work [35] proposes GUI reverse engineering for Android based on a combination of static code analysis and run-time analysis. Such an approach is more effective than purely-dynamic GUI ripping, and it presents a good starting point for our testing technique.

III. Test Amplification Approach

The starting point of our approach is a set of test cases T based on the GUI model of the application. The choice of this set is orthogonal to the amplification approach we propose. In our current implementation we use a set T that covers each state and each transition in the GUI model at least once. Each test case $t \in T$ is implemented in the Robotium testing framework [30], where calls to the framework API trigger the necessary GUI events. Note that alternative definitions for T are also possible (e.g., to achieve coverage of certain subpaths in the GUI model), and they present an interesting target for future work.

One simple approach to detect potentially-problematic operations is to execute each t and to observe (and report) any suspicious calls occurring during test execution. In fact, the Android platform already provides a so-called "strict mode" [31], which reports if any calls to a hard-coded set of expensive APIs (e.g., `java.net.*`) occur at run time. Alternatively, code instrumentation at such suspicious calls could be inserted by the tester (e.g., using an instrumentation tool) to report the run-time occurrences of these calls.

We propose a more comprehensive approach which demonstrates and quantifies the direct cause-effect relationship between expensive operations and poor responsiveness observed by the user. In this approach, each test case $t \in T$ is first executed against the unmodified application, and the response time for each GUI event is recorded. Next, the execution of each t is *amplified* according to one of several amplification criteria. The use of the term "amplification" follows earlier work [39], where test cases are amplified by throwing additional exceptions at API calls, to test robustness in the presence of exceptional behavior. Our approach amplifies relevant API calls with artificial delays, re-executes t, and measures the new GUI event response times. The changes in response times, as a function of the duration of the delays at API calls, provides a characterization of the effects of these calls on the user. For example, if we want to focus on a particular API call site c, we would instrument c (and only c) with a configurable delay d, then execute each $t \in T$ for several different values of d, and report the changes in GUI event response times. The effect of c on the user is fully characterized by the relationship between the values of d, the values of response times, and the specific GUI events for which responsiveness decreases.

A. Amplification Criteria

Based on Android developer guidelines [20], [10], [31], as well as our own experience, we focus on four categories of API calls that represent the common sources of jank. These APIs are related to sending and receiving data over the network, accessing the flash storage, accessing the on-device database, and processing of bitmaps.

1) Network Access: A well-known development guideline states that Android applications should not perform network operations in the UI thread [20]. Even a small HTTP fetch over a 3G network can take several seconds [10], far exceeding the limit of 200 ms needed to avoid the perception of sluggishness. Our experience shows that a variety of API calls can be used to access the network (e.g., using class `org.apache.http.client.HttpClient`, but we have observed a number of other similar APIs). Currently we use automated heuristics together with manual steps to identify such calls; in total, 15 network-related API methods are considered. An interesting direction for future work is to perform such identification fully automatically, with the help of either static or dynamic program analysis.

2) Flash Storage Access: The file system in Android devices is implemented on top of flash storage.[1] The performance of the file system presents some challenges: for example, the degree of concurrency for multiple disk operations (from different simultaneously-running applications) may be low, and the cost of the disk garbage collection may be high. The

[1]Android defines two categories of non-volatile storage: internal and external, possibly accessed through different file systems such as `YAFFS` and `exFAT`. Both categories are implemented with flash memory.

general guideline is that even simple disk operations could exhibit significant and unexpected latencies, and should be avoided in the UI thread [10]. Our approach amplifies disk-access calls to 25 methods in relevant stream classes.

3) Database Access: Android applications often access the on-device SQLite database by calling methods in class `android.database.sqlite.SQLiteDatabase`. The database accesses can generate substantial amount of expensive write operations to the flash storage. Furthermore, depending on the query, the effectiveness of the SQLite query optimizer may differ significantly [10]. We amplify calls to 11 methods in `SQLiteDatabase`, including `delete`, `execSQL`, `insert`, `query`, `rawQuery`, `replace`, and `update`.

4) Bitmap Processing: Processing of large bitmaps could be computationally expensive and should not be done in the UI thread [20]. A call to a method `BitmapFactory.decode*` will load and decode an image from the file system, potentially producing a smaller subsampled version in memory [21]. We amplify calls to 6 such methods, to simulate a situation when a large HD image from disk is loaded and decoded, which could "freeze" the application's user interface and potentially trigger an ANR error.

B. Implementation and Test Execution

Our implementation of the amplification uses AspectJ to instrument the application. This requires access to the source code. The approach can also be applied to programs for which source code is not available, either by (1) instrumenting the Android-specific Dalvik VM bytecode directly [29], [17], or (2) converting Dalvik bytecode back to Java bytecode [27] and using existing bytecode rewriting tools.

Instrumentation is added immediately before each of the API calls described earlier. Each instrumented call site c is given a unique ID. Immediately before test cases are executed, a particular call site ID is activated (the rest of the IDs remain inactive). During test execution, the instrumentation at the activated call site c introduces a configurable delay to the execution of the current thread. The duration of the delay can be varied from run to run, and the GUI event response times can be used to quantify the effect on responsiveness. In cases where the goal is simply to trigger an ANR error for c, we introduce a delay of 10 seconds at c, run all test cases, and determine whether at least one fails with an ANR error. Such failures can be detected automatically by considering the contents of an ANR log file maintained by the Android platform. In our experience, the ANR errors triggered in this manner occur deterministically.

IV. Evaluation

We performed an initial study of the proposed testing approach on eight open-source Android applications. For each application we constructed a set of test cases that cover all GUI states and transitions. All experiments were performed on the standard Android emulator from the Android SDK.

The characteristics of the experimental subjects are shown in the first five columns of Table I. The applications come from several domains: email client (K9), SSH client (ConnectBot), PDF and e-book readers (APV, VuDroid, FBReader), task management (astrid), password management (KeePassDroid), and multimedia player (VLC). The table shows the number of activities (each one corresponding to different state in the GUI model) and the number of Java classes in each application. Column "Test Cases" shows the number test cases that were used as input to the amplification process.

Each subsequent column in Table I is divided into two subcolumns. The first subcolumn shows the number of all API call sites that were subjected to a particular amplification criterion, as described in Section III-A. The second subcolumn shows how many of these call sites, when amplified individually with a large delay, triggered an ANR error failure for at least one test case.

The measurements in Table I demonstrate that each of the eight analyzed applications violates the best-practice guidelines defined by Google engineers [20], [10] and contains janky operations in the UI thread. The total number of such operations is 61. It is interesting to note the developers of these applications appear to be quite careful in their handling of network operations: since such operations are widely-known to be harmful to responsiveness, developers do not include them in the UI thread (only one ANR-triggering operation was related to the network). The other sources of jank are more prevalent, with database accesses being the most common ones. These observations indicate that both jank prevention (e.g., via code analysis tools during software development) and jank detection (e.g., as part of software testing) are important consideration for Android, and that the full diversity of root causes should be taken into account by future analysis and testing techniques.

V. Case Studies

This section presents four case studies based on the failing test cases we observed. An interesting question raised by these studies is how to proactively prevent such responsiveness problems, through design principles and patterns, and with the help of automated code transformation techniques.

A. Connectbot

Connectbot is an SSH client for Android. For this application the amplified test cases highlighted an API call which is used to send keyboard input to a remote machine. Whenever the user presses a button on the keyboard, the button information is encrypted and sent over the network. Depending on network congestion and the status of the remote machine, the send operation (a write on a socket) may block the UI thread for several seconds. As a result, the application would become unresponsive and an ANR error would occur. It may be desirable to redesign the application to perform the network send in a separate thread, without "freezing" the UI thread.

B. K-9 Mail Client

K-9 is a popular open-source email client for Android. For this application we found a problem related to flash

TABLE I
EXPERIMENTAL RESULTS.

Application	Version	Activities	Classes	Test Cases	Network		Flash		Database		Bitmap	
					All	Fail	All	Fail	All	Fail	All	Fail
APV	r131	4	56	13	0	0	5	1	4	1	0	0
astrid	cb66457	11	481	18	2	0	11	0	62	3	0	0
ConnectBot	e63ffdd	9	301	14	6	1	5	1	23	14	0	0
FBReader	a53ed81	22	757	17	2	0	22	4	43	8	2	1
KeePassDroid	085f2d8	7	126	21	0	0	7	0	14	6	0	0
K9	v0.114	15	418	14	20	0	25	1	41	9	3	0
VLC	dd3d61f	8	176	18	1	0	8	1	22	3	8	1
VuDroid	r51	3	67	11	0	0	2	2	0	0	4	4

storage accesses. The problem occurs when an user attempts to download an email attachments to the local file system. When the user clicks on the download button, the attachment file(s) are written to flash storage by the UI thread. Android provides alternative mechanisms to perform such a potentially-expensive task, including class `android.os.AsyncTask` which can be used to perform background operations that publish results on the UI thread.

C. VLC Media Player

VLC is a cross-platform multimedia player designed to decode and play video/audio files from local storage and from a network stream. In this application, we found a problem related to decoding an image from external storage into a bitmap object. Before VLC plays any audio file, it uses the UI thread to load the cover page of the album. This image, in most cases, is stored in flash storage. The decoding of a large bitmap may result in poor responsiveness. Furthermore, the same image could be decoded multiple times if the user switches back and forth between different audio files.

D. Astrid

Astrid is an assistant-like application which helps users manage their schedules by adding, deleting, searching, and modifying tasks. In this application we found several occurrences where the UI thread performs SQLite database operations when the user manipulates her tasks. As described in Section III-A, the performance of database operations depends on a variety of factors, and can cause poor responsiveness. To avoid this problem, a number of other Android mechanisms could be used, including `android.os.Handler`, `android.os.AsyncTask`, and `android.app.IntentService`.

VI. RELATED WORK

Model-based GUI testing. Model-based GUI testing (e.g., [33], [24], [25], [22], [34], [32], [1], [14]) builds a finite state machine model for the application GUI and generates test cases based on this model with respect to various coverage criteria (e.g., [24]). The focus of all these existing approaches is on functional correctness. We are interested in non-functional properties related to application responsiveness. The proposed amplification technique and criteria consider important resources in the Android platform, and detect ANR problems in the presence of variations in response times for GUI events that access these resources. As an alternative, random testing can also be applied in testing of GUI applications. For example, work in [16] uses the Monkey tool [26] to randomly insert GUI events into a running Android application, and then analyzes the execution log to detect faults. Random testing is highly unlikely to uncover responsiveness defects in the target application.

To build a GUI model, dynamic reverse engineering is typically employed. GUI ripping [23], [25] automatically traverses at run time the application GUI to extract a corresponding finite state machine model. AndroidRipper [1] applies this approach to Android applications. Several techniques have been proposed to improve the precision of models and the test cases generated from them (e.g., [37], [38], [36], [13], [5], [14]). Recent work [35] proposes a static analysis which can be combined with dynamic reverse engineering to create better GUI models for Android applications and to improve model-based testing.

Measuring responsiveness. Due to its negative effects on user satisfaction, poor responsiveness in desktop GUI applications has been the focus of several existing techniques (e.g., [9], [19], [18]). In all these approaches, measurement techniques are proposed to characterize and understand the latency of event handling code. LagHunter [18] uses several heuristics to select a set of methods to be tracked, and measures only the running times of the selected methods in order to identify lags in the handling of user events. This techniques is applied to Java applications that use the Swing or SWT GUI toolkits. In contrast, our approach measures GUI event response times in Android software, both for the original program as well as for an amplified program. The differences in response times are used to characterize the effect of a suspicious API call site on the user-observed application responsiveness. Furthermore, our approach systematically explores the space of GUI events and code operations that may cause poor responsiveness, while LagHunter profiles a given run-time execution and can only report suspicious behavior that was observed for this particular execution.

Testing and analysis for Android. Anand et al. [2] use symbolic execution for generation of Android event sequences to be used in testing. Zhang and Elbaum [39] apply amplification techniques for testing of exception-handling code when

accessing unreliable resources. There is emerging body of work on analysis of Android applications—for example, for security (e.g., [6], [7], [17]), privacy (e.g., [8]), and energy (e.g., [28], [15]).

VII. Conclusions and Future Work

Poor responsiveness of Android software can be very harmful to user perception and marketplace success. We propose a test amplification approach that exposes and quantifies the root causes of responsiveness defects. Our promising initial study suggests that such defects occur regularly in Android applications, and that the proposed technique is highly effective in discovering them.

Mobile applications are increasingly important both for software users and for software researchers. The quality of such applications can and should be improved with the help of automated testing and analysis. Our experience points to a number of interesting research problems in this domain, including (1) automated discovery (via static and/or dynamic analysis) of code that manipulates expensive resources, (2) better reverse engineering of GUI models, (3) model-based generation of more comprehensive tests to be used as input to amplification, (4) static analyses to detect responsiveness problems before testing, and (5) design principles and patterns to proactively prevent responsiveness defects.

Acknowledgments

We thank the MOBS reviewers for their valuable comments and feedback. This material is based upon work supported by the U.S. National Science Foundation under grant CCF-1017204 and by a Google Faculty Research Award.

References

[1] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using GUI ripping for automated testing of Android applications. In *ASE*, pages 258–261, 2012.

[2] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *FSE*, pages 1–11, 2012.

[3] Android GUITAR. `sourceforge.net/apps/mediawiki/guitar/index.php?title=Android_Guitar`.

[4] APV PDF viewer. `code.google.com/p/apv`.

[5] S. Arlt, A. Podelski, C. Bertolini, M. Schäf, I. Banerjee, and A. M. Memon. Lightweight static analysis for GUI testing. In *ISSRE*, 2012.

[6] A. Chaudhuri. Language-based security on Android. In *PLAS*, pages 1–7, 2009.

[7] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *MobiSys*, pages 239–252, 2011.

[8] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, pages 1–6, 2010.

[9] Y. Endo, Z. Wang, J. B. Chen, and M. Seltzer. Using latency to evaluate interactive system performance. In *OSDI*, pages 185–199, 1996.

[10] B. Fitzpatrick. Writing zippy Android apps. In *Google I/O Developers Conference*, 2010.

[11] Gartner, Inc. Press release, 2012. `www.gartner.com/newsroom/id/2237315`.

[12] Google Play app store. `play.google.com/store/apps`.

[13] R. Gove and J. Faytong. Identifying infeasible GUI test cases using support vector machines and induced grammars. In *TESTBED*, pages 202–211, 2011.

[14] F. Gross, G. Fraser, and A. Zeller. Search-based system testing: High coverage, no false alarms. In *ISSTA*, pages 67–77, 2012.

[15] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *ICSE*, 2013.

[16] C. Hu and I. Neamtiu. Automating GUI testing for Android applications. In *AST*, pages 77–83, 2011.

[17] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. Android and Mr. Hide: Fine-grained permissions in Android applications. In *SPSM*, 2012.

[18] M. Jovic, A. Adamoli, and M. Hauswirth. Catch me if you can: Performance bug detection in the wild. In *OOPSLA*, pages 155–170, 2011.

[19] M. Jovic and M. Hauswirth. Listener latency profiling: Measuring the perceptible performance of interactive Java applications. *Science of Computer Programming*, 76(11):1054–1072, 2011.

[20] Keeping your app responsive. `developer.android.com/training/articles/perf-anr.html`.

[21] Loading large bitmaps efficiently. `developer.android.com/training/displaying-bitmaps/load-bitmap.html`.

[22] A. M. Memon. An event-flow model of GUI-based applications for testing. *Software Testing, Verification and Reliability*, 17(3):137–157, 2007.

[23] A. M. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *WCRE*, pages 260–269, 2003.

[24] A. M. Memon, M. L. Soffa, and M. E. Pollack. Coverage criteria for GUI testing. In *FSE*, pages 256–267, 2001.

[25] A. M. Memon and Q. Xie. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *TSE*, 31(10):884–896, 2005.

[26] Monkey: UI/Application exerciser for Android. `developer.android.com/tools/help/monkey.html`.

[27] D. Octeau, S. Jha, and P. McDaniel. Retargeting Android applications to Java bytecode. In *FSE*, page 6, 2012.

[28] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps. In *MobiSys*, pages 267–280, 2012.

[29] Redexer: A Dalvik bytecode instrumentation framework. `www.cs.umd.edu/projects/PL/redexer`.

[30] Robotium testing framework for Android. `code.google.com/p/robotium`.

[31] Strict mode in Android. `developer.android.com/reference/android/os/StrictMode.html`.

[32] T. Takala, M. Katara, and J. Harty. Experiences of system-level model-based GUI testing of an Android application. In *ICST*, pages 377–386, 2011.

[33] L. White and H. Almezen. Generating test cases for GUI responsibilities using complete interaction sequences. In *ISSRE*, pages 110–121, 2000.

[34] Q. Xie and A. M. Memon. Using a pilot study to derive a GUI model for automated testing. *TOSEM*, 18(2):7:1–7:35, 2008.

[35] W. Yang, M. Prasad, and T. Xie. A grey-box approach for automated GUI-model generation of mobile applications. In *FASE*, 2013.

[36] X. Yuan, M. B. Cohen, and A. M. Memon. GUI interaction testing: Incorporating event context. *TSE*, 37(4):559–574, 2011.

[37] X. Yuan and A. M. Memon. Using GUI run-time state as feedback to generate test cases. In *ICSE*, pages 396–405, 2007.

[38] X. Yuan and A. M. Memon. Generating event sequence-based test cases using GUI run-time state feedback. *TSE*, 36(1):81–95, 2010.

[39] P. Zhang and S. Elbaum. Amplifying tests to validate exception handling code. In *ICSE*, pages 595–605, 2012.