Generating various contexts from permissions for testing Android applications

Kwangsik Song, Ah-Rim Han, Sehun Jeong, Sungdeok Cha
Department of Computer Science and Engineering
Korea University
Seoul, South Korea
{kwangsik_song, arhan, gifaranga, scha}@korea.ac.kr

***Abstract*—Context-awareness of mobile applications yields several issues for testing, since the mobile applications should be testable in any environment and with any contextual input. In previous studies of testing for Android applications as event-driven systems, many researchers have focused on using the generated test cases considering only GUI events. However, it is difficult to detect failures in the changes in the context in which applications run. It is important to consider various contexts since the mobile applications adapt and use novel features and sensors of mobile devices. In this paper, we provide the method of systematically generating various executing contexts from permissions. By referring the lists of permissions, the resources that the applications use for running Android applications can be inferred easily. The various contexts of an application can be generated by permuting resource conditions, and the permutations of the contexts are prioritized. We have evaluated the usefulness and effectiveness of our method by showing that our method contributes to detect faults.**

Keywords*—Android application testing, permissions, various contexts, context-aware application, mobile application testing

I. INTRODUCTION

The proliferation of the novel features and sensors of mobile devices (i.e., operating systems, hardware platforms, and device sensors) has enabled the development of mobile applications that can provide rich, highly-localized, context-aware content to users [1]. In particular, the market for disease diagnostic systems is growing fast due to the development of mobile applications that log personal health data (e.g., blood glucose, blood pressure, and heart rate) by using the sensors, cameras, additional simple adapters (or accessories) in mobile devices and sending the results to the system in real-time. For instance, in the mobile application called *Peek Vision* [2], medical images can be captured by using a clip-on camera adapter that gives high quality images of the back of the eye and can be sent to the system so diagnosis can be done remotely. The mobile application has been designed to be aware of the computing context in which it runs and to adapt and react according to its findings; therefore, it belongs to the category of context-aware applications [3].

The context-awareness of mobile applications yields several issues for testing [4] because the mobile applications should be testable in any environment and with any contextual input [5]. These applications are notified of a change to their context by means of events, and the variability in the running conditions of a mobile application depends on the possibility of using it in variable contexts. A context represents the overall environment that the application is able to perceive [6]. More precisely, Abowd et al. [7] define a context as: "any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is including the user and applications themselves."

In previous studies of testing of *Android applications* as event-driven systems, many researchers focused on using the generated test cases considering only GUI events. However, it is difficult to detect failures in the changes in the contexts, which can be influenced by context events, in which applications run. Even in studies that considered context events, the specific event sequences generated based on a limited number of scenarios were considered. This has limitations in terms of finding bugs that occur in various complex contexts. It is important to discover the unacceptable behaviors of an app (such as crashes or freezes) that are often reported in the bug reports of mobile apps and appear when the app is impulsively solicited by contextual events, such as the alerts for the connection/disconnection of a plug (e.g., USB and headphone), an incoming phone call, GPS signal loss, etc. Therefore, for testing mobile applications, we need a systematic testing method to take into account the various conditions in context-aware systems. This is increasingly needed given trends in mobile applications due to the advancement in the novel features and sensors of mobile devices, which reveal new types of bugs [8].

To access resources from Android devices, each Android application includes a manifest file, `AndroidManifest.xml`, which lists the permissions [9] that the application requires for its execution and requests permissions for the resources. By referring to the lists of permissions, the resources that the applications use for running Android applications can be inferred easily. The context events occurred from/by those identified resources, and the state for each condition can be changed by those context events. Thus, we use the permissions to generate the various contexts used for testing Android applications.

To test mobile applications in various contexts, we provide a method for systematically generating **various executing contexts** from permissions. In our paper, an executing context represents a permutation of resource conditions that have variable states, and Graphical User Interface (GUI) event based generated test cases [10] can be run in those contexts. The state of each condition can be changed/sensed/perceived according

DOI reference number: 10.18293/SEKE2015-118

to several types of context events, such as:

- events coming from the **external environment** and sensed by device sensors (e.g., Wi-Fi and GPS);
- events generated by the **device hardware** platform (e.g., battery and other external peripheral port, such as USB, headphone, and network receiver/sender); and
- events typical of mobile phones (e.g., the arrival of a phone call or a SMS message).

The brief procedure for generating various executing contexts using permissions is as follows. First, the related resources and their possible states are identified from the permissions. Then, the various executing contexts are generated by permuting the resource conditions that have variable states. Finally, the executing contexts are prioritized and the part of those executing contexts are selected. We applied our testing method to two open-source projects, Open Camera [11] and Subsonic [12]. Experiments reveal that the proposed method is significantly effective in detecting faults.

The rest of this paper is organized as follows: Section II contains a discussion of related studies. Section III explains the definition and the need to use permissions when testing Android applications, and the related resources that can inferred from the permissions are identified. Section IV explains the procedure to generate various contexts from the permissions. In Section V, we present an experiment to evaluate the proposed approach and discuss the results. We conclude and discuss future research in Section VI.

II. Related Work

Mobile applications are event-driven systems, but, unlike other traditional event-driven software systems, GUI [10], [13]–[15] or web applications [16], they are able to sense and react to a wide range of events. In the following subsections, we discuss the related studies that provide methods for testing *Android applications* as event-driven systems.

A. GUI Testing

Random testing. The UI/Application Exerciser Monkey [13] is part of the Android SDK and generates random user input. Originally designed for stress-testing Android applications, it randomly generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. Monkey testing is a random and automated unit test. The test is not scripted and is run mainly to check whether a system or an application will crash. It is easy to set up and can be used in any application. The cost of using the testing is relatively small. However, detection of only a few bugs is possible.

Model-based testing. *AndroidRipper* [14] is an automated technique implemented in a tool that tests Android applications using a GUI model. AndroidRipper is based on a user interface-driven ripper that automatically explores the application's GUI to exercise the application in a structured manner. More specifically, it dynamically analyses the application's GUI for obtaining sequences of events that are *fireable* through the GUI widgets. Each sequence provides an executable test case. During its operation, AndroidRipper maintains a state machine model of the GUI (called a GUI Tree). The GUI Tree contains the set of GUI states and state transitions encountered during the ripping process. However, by using generated test cases that consider only GUI events, it is difficult to find failures that could otherwise be detected by considering the changes in the context, which can be influenced by context events, in which applications run.

B. Context-aware Testing

Amalfitano et al. [6] took into account both context events and GUI events for testing Android applications. They manually define reusable *event patterns*—representations of event sequences that abstract meaningful test scenarios. These event-patterns are manually defined after a preliminary analysis is conducted on the bug reports of open source applications. Based on the defined event patterns, test cases are generated using the three scenario-based mobile testing approaches that (1) manually generate test cases, (2) mutate existing test cases, and (3) support the systematic exploration of the behavior of an application (an extension of the GUI ripping technique is presented in [14]). For dynamically recognizing the context events that the application is able to sense and react at a given time, events can be deduced from event handlers. In this work, they also use a set of Intent Messages to figure out the events that are managed by other application components. This set can be obtained by means of static analysis of the a Android manifest file of the application.

The methodology proposed by Amalfitano et al. has some limitations. The number of scenarios that define relevant ways of exercising an application is limited because specific event sequences are considered. By manual analysis by experts, the events possibly trigger a faulty behavior may not be properly identified. By analyzing bug history, a sequence of events that has never occurred might not be chosen, but they may cause catastrophic failures. These event patterns may need to be redefined when testing other types of applications. From the perspective of triggering the context events, the source codes also need to be analyzed and altered. Moreover, the effectiveness of the testing approach is evaluated only by measuring the code coverage. Statement coverage may not be effective and sufficient enough on fault detection capability [17]. In our paper, we provide a systematic method of generating various executing contexts. Since this method may cause to produce many test cases to be run, we provide a prioritization technique to rank the test cases in the order of the likelihood of detecting faults.

III. Inferring Resources from Permissions

A. Permissions in Android Application

Android uses a system of permissions [9] to control how applications access sensitive devices and data stores. More specifically, to ensure security and privacy, Android uses a permission-based security model to mediate access to sensitive data (e.g., location, phone call logs, contacts, emails, or photos) and potentially dangerous device functionalities (e.g., Internet, GPS, and camera) [18].

To access resources from Android devices, each Android app requests permissions for resources by listing the permissions. Each Android application includes a manifest file,

TABLE I: List of permissions and related resources with their possible states.

Permission	Allows an App to	Related Resources[Possible States]	Android Version
ACCESS_FINE_LOCATION	Access precise location from location sources	Wi-Fi[on|off], GPS[on|off], Radio[on|off]	Android 1.0 ~
INTERNET	Open network sockets	Wi-Fi[on|off], Radio[on|off]	Android 1.0 ~
CAMERA	Be able to access the camera device	Camera [on|off], SD card[free|full]	Android 1.0 ~
BLUETOOTH	Connect to paired bluetooth devices	Bluetooth[on|off]	Android 1.0 ~
WRITE_CALL_LOG	Write (but not read) the user's contacts data	Radio[on|off]	Android 4.0.3 ~
WRITE_EXTERNAL_STORAGE	Write to external storage	SD card[on|off]	Android 1.5 ~
BIND_DEVICE_ADMIN	Ensure that only the system can interact with device	Camera [on|off], Flash[on|off], SD card[free|full], Wi-Fi[on|off]	Android 2.2.x ~
VIBRATE	Access to the vibrator	Vibrator[on|off]	Android 1.0 ~
NFC	Perform I/O operations over NFC	NFC[on|off]	Android 2.3 ~
FLASHLIGHT	Access to the flashlight	Flash[on|off]	Android 1.0 ~
CHANGE_NETWORK_STATE	Change network connectivity state	Wi-Fi[on|off], GPS[on|off], Radio[on|off]	Android 1.0 ~
CAPTURE_VIDEO_OUTPUT	Capture video output	LCD[on|off], Camera [on|off]	Android 1.0 ~

`AndroidManifest.xml` [19], which lists the permissions that the application requires for its execution. When the user wants to install an app, this list of permissions is presented and confirmation is requested. When the user confirms the access, the app will have the requested permissions at all times (until the app is uninstalled). If an application requests the resource without having the appropriate permission, then the Android OS may throw a Security Exception or simply not grant the requested resource [20]. These permission-protected resources are accessed through the Android API and other classes resident on the phone. For example, having the *ACCESS_FINE_LOCATION* permission will give the application access to a number of Android API calls that use resources such as GPS, Wi-Fi, and Radio.

B. *Identification of Related Resources from Permissions*

We have used the permissions in an app's manifest file for generating various context used for testing Android applications. By referring to the lists of permissions, the resources that the applications would (potentially) use for running Android applications can be inferred easily. without analyzing source codes of the applications. The context events occur from/by those identified resources, and the state for each condition can be changed by those context events. Thus, by using permissions, we can generate various executing contexts that represent permutations of resource conditions that have variable states.

The latest Android platform release contains a list of 152 permissions. Among them, we focus on the permissions related to communicating with the environments, because they are more critical for making context-aware apps. For each permission, the related resources with their possible states are identified in Table I. It is intuitive to identify the related resources in the permissions of *BLUETOOTH* or *CAMERA*. Meanwhile, in the permission of *ACCESS_FINE_LOCATION*, it covers multiple resources such as GPS, Wi-Fi, and Radio. To consider the variable states of resource conditions, the possible states are defined in terms of an availability (i.e., on or off). It is also worth to note that the table is independent to the features of an app and thus it is reusable.

IV. Testing Android Applications in Various Contexts

Fig. 1 represents the overall procedure for generating various executing contexts using permissions.

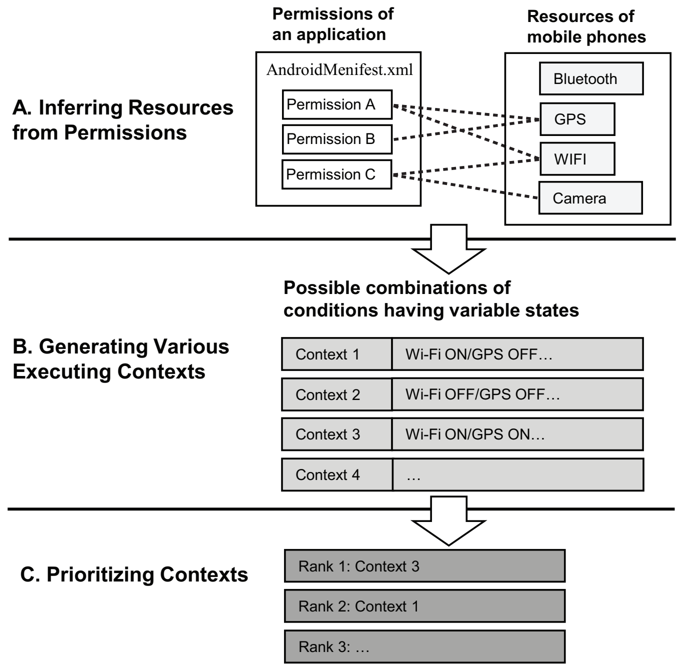


Fig. 1: An overview for generating various contexts after analyzing permissions.

A. *Generating Various Executing Contexts*

The executing contexts of an app can be generated by permuting resource conditions. For instance, if the resources that an app uses are r_1, r_2, ..., r_n, and the number of possible states for those corresponding resource conditions are $N(r_1)$, $N(r_2)$, ..., $N(r_n)$, then the total number of generated executing contexts is $N(r_1) \times N(r_2) \times ... \times N(r_n)$. For example, if an app's permission has links with Bluetooth, GPS, and Wi-Fi, then net executing contexts include eight different permutations because each resource condition has two candidate states.

B. *Prioritizing Contexts*

While the generation of executing contexts is straightforward and easy to automate, the number of generated executing contexts increases as the number of considered resources increase. An app is executed on every test case for all the

generated various contexts, and the test runs increase exponentially. Thus, we need to prioritize the executing contexts to select the contexts to be tested first.

We suggest the two-level prioritizing strategies to rank the generated executing contexts. The first step is weighting each resource condition according to the testing objectives (e.g., testing normal or unacceptable behaviors). To test normal behaviors of the apps, the executing contexts, in which more resources are used, should be more highly ranked. Thus, for example, weights can be assigned to resource conditions as follows: Wi-Fi[on]=1, GPS[on]=1, Camera[on]=1, and SD card[free]=1. If the objective of the testing is to detect unacceptable behaviors of an app, then executing contexts related to the exceptional scenarios should be more highly ranked; thus, the resource conditions constituting those executing contexts need to be weighted, such as Wi-Fi[off]=1, GPS[off]=1, Camera[off]=1, and SD card[full]=1. To obtain the score of each generated executing context, the weights of resource conditions of the executing context are summed.

In the second step, to distinguish the executing contexts that have the same scores, we provide the method to assign weights to individual or combinatorial resources residing in an executing context. We suggest three criteria—frequency, user controllability, and minimum required resource conditions—as follows.

- **Frequency.** It represents how much a resource is required via permissions and is to be used in an app. It counts the identified number of each resource over the lists of permissions. For example, let an app have the permissions in Table I, then the frequency of the Radio resource is four. Thus, frequently identified resources need can be weighted to test more used resource-related behaviors of an app first.
- **User controllability.** It indicates how easily a user can control a resource. For example, users can enable or disable GPS or Wi-Fi but do not control hardware-related sensors directly. Thus, resources that are more user controllable can be weighted to test usable resource-related behaviors of an app first.
- **Minimum required resource conditions.** The certain combinations of resource conditions need to be weighted to test permission-related behaviors first. The rational of the idea comes from the observations that several permissions are related to multiple resources and require minimum resource conditions to provide expected services to an app. For example, the *ACCESS_FINE_LOCATION* permission uses three resources, GPS, Wi-Fi, and Radio; and among the three resources, GPS[on] and Wi-Fi[on] are the *necessary and sufficient* resource conditions to provide the service which is to access precise location from location sources. On the other hand, if we focus on detecting faults, the combination of states that could trigger a faulty behavior (e.g., GPS[on] and Wi-Fi[off]) could be more highly weighted.

V. Evaluation

We investigated the two research questions in our experiment.

TABLE II: Characteristics for each experimental subject.

Name	**Open Camera (Ver. 1.21)** [11]	**Subsonic for Android (Ver. 4.4)** [12]
Description	Taking pictures and providing various features (e.g., zooming, focusing, flashing, and coloring effects)	Playing music and video by receiving media files from the stream server (e.g., personal PC) and supports offline mode and bitrates
Class #	61	265
Method #	399	1038
LOC #	3,790	16,064

TABLE IV: Bugs that can be detected using our approach.

Open Camera		Subnonic	
Fault No.	**Bug ID. (refer in [21])**	**Fault No.**	**Bug ID. (refer in [22])**
1	1	1	150
2	2	2	126
3	9	3	64
4	20	4	102
5	3	5	38
6	11	6	82
7	30	7	46
8	31	8	39
9	37	9	35
10	4	10	32
11	28	11	21
12	33	12	8
		13	4
		14	83

RQ1. Is our testing approach useful for detecting faults?

RQ2. Is our prioritization technique effective in detecting faults?

A. Experimental Design

Two open source projects are chosen as experimental subjects: Open Camera [11] and Subsonic [12]. We selected these as subjects because they are open source projects and their development histories (such as bug issues) are accessible. They contain a relatively large number of classes and methods (large size) as well. Table II summarizes characteristics of each subject.

The testing is performed by running the test cases under each context. In other words, the same test cases had run in a iterative manner as much as the number of the (selected) contexts. We first execute test cases generated from the Android GUI ripper tool [10]. They provide the sequences of events associated with GUI tree paths that link the root node to the leaves of the tree, but the results of statement code coverage on the experimental subjects were low (i.e., average from 45% to 47%). Since GUI-based approaches have limitations for covering all components, we additionally performed testing by focusing on the scenarios that users use more frequently and faulty behaviors may be more occurred.

Table III shows the generated and used executing contexts for Open Camera and Subsonic. As mentioned in Section IV-A, the executing contexts to be tested first need to be prioritized and selected because too many test runs are required, which is computation-intensive. To test the normal scenario, we select the context where all resources are on (active). On the other hand, to test the exceptional scenario, we also select the context where all resources are off (inactive). The contexts that might

TABLE III: Generated and used executing contexts from our approach.

<table>
<tr><th rowspan="2">Name</th><th colspan="4">Executing Contexts</th></tr>
<tr><th>Permission</th><th>Resource[States]</th><th>Total #</th><th>Used #</th></tr>
<tr><td rowspan="3">Open Camera [11]</td><td>ACCESS_FINE_LOCATION</td><td>Wi-Fi[on|off], GPS[on|off], Radio[on|off]</td><td rowspan="3">32 $= 2^5$</td><td rowspan="3">
<table>
<tr><th>Rank</th><th>Wi-Fi</th><th>GPS</th><th>Radio</th><th>SD card</th><th>Camera</th></tr>
<tr><td>1</td><td>on</td><td>on</td><td>on</td><td>free</td><td>enable</td></tr>
<tr><td>2</td><td>off</td><td>off</td><td>off</td><td>full</td><td>disable</td></tr>
<tr><td>3</td><td>on</td><td>on</td><td>on</td><td>full</td><td>enable</td></tr>
<tr><td>4</td><td>off</td><td>on</td><td>off</td><td>free</td><td>enable</td></tr>
</table>
</td></tr>
<tr><td>CAMERA</td><td>Camera [on|off], SD card[free|full]</td></tr>
<tr><td>WRITE_EXTERNAL_STORAGE</td><td>SD card[free|full]</td></tr>
<tr><td rowspan="9">Subsonic [12]</td><td>INTERNET</td><td>Wi-Fi[on|off], Radio[on|off]</td><td rowspan="9">128 $= 2^7$</td><td rowspan="9">
<table>
<tr><th>Rank</th><th>Wi-Fi</th><th>Radio</th><th>Bluetooth</th><th>SD card</th><th>Audio</th><th>MIC</th><th>CPU</th></tr>
<tr><td>1</td><td>on</td><td>on</td><td>on</td><td>free</td><td>enable</td><td>on</td><td>unlock</td></tr>
<tr><td>2</td><td>off</td><td>off</td><td>off</td><td>full</td><td>disable</td><td>off</td><td>lock</td></tr>
<tr><td>3</td><td>on</td><td>on</td><td>off</td><td>free</td><td>enable</td><td>on</td><td>unlock</td></tr>
<tr><td>4</td><td>on</td><td>on</td><td>on</td><td>full</td><td>enable</td><td>on</td><td>unlock</td></tr>
<tr><td>5</td><td>on</td><td>on</td><td>on</td><td>free</td><td>enable</td><td>on</td><td>lock</td></tr>
<tr><td>6</td><td>off</td><td>on</td><td>on</td><td>free</td><td>enable</td><td>on</td><td>unlock</td></tr>
</table>
</td></tr>
<tr><td>BLUETOOTH</td><td>Bluetooth [on|off]</td></tr>
<tr><td>RECORD_AUDIO</td><td>Audio[on|off], MIC[on|off]</td></tr>
<tr><td>READ_PHONE_STATE</td><td>Radio[on|off]</td></tr>
<tr><td>WRITE_EXTERNAL_STORAGE</td><td>SD card[free|full]</td></tr>
<tr><td>WAKE_LOCK</td><td>CPU[lock|unlock]</td></tr>
<tr><td>MODIFY_AUDIO_SETTINGS</td><td>Audio[on|off]</td></tr>
<tr><td>ACCESS_NETWORK_STATE</td><td>Wi-Fi[on|off], Radio[on|off]</td></tr>
<tr><td>READ_EXTERNAL_STORAGE</td><td>SD card[free|full]</td></tr>
</table>

cause faulty behavior are also highly ranked. For example, the faults may be more occurred on the situations when SD card is full (it many cause a problem in file processing) and when GPS is on while Wi-Fi is off (it may result in logging wrong location information). When setting these contexts, other resources—not involving these situations—are set to be inactivated. As a result, we selected four among 32 and six among 128 executing contexts in Open Camera and Subsonic, respectively.

Since the subjects are open source projects, we can access the bug histories of Open Camera [21] and Subsonic [22]. We manually analyzed the issues in those repositories and extracted the faults that could have been detected if our testing technique had been used. Then, we execute the test cases in the contexts generated using our approach and discover unacceptable behaviors of the app (such as crashes or freezes). These bugs are matched the corresponding faults extracted from the repositories. For example, we found a crash (i.e., runtime exception: fail to connect camera service) by executing test cases in the context where a camera is disabled. This crash can be matched to the faults of camera malfunctions or exceptions.

To evaluate the effectiveness of our method of prioritizing contexts, we compare three different sequences of contexts in which test cases run: the generated order T, the reversed order T_r, and the prioritized order T_p (which is ranked according to our prioritization method). The order T is the order generated from our approach but not prioritized. The generated order T and the reversed order T_r can be regarded as random sequences.

To quantify the capability of the contexts on fault detection, we use a metric called APFD (Average Percent Fault Detection) [23]. The APFD is calculated by taking the weighted average of the percentage of faults detected over the life of the suite. The higher numbers imply faster (better) fault detection rate. Let T be a test suite containing n contexts in which test cases run, and let F be a set of m faults revealed by T. Let TF_i be the fist context in ordering of T' of T which reveals fault i. The APFD for test suite T' is given by the equation:

$$APFD = 1 - \frac{TF_1 + TF_2 + ... + TF_m}{nm} + \frac{1}{2n}.$$

We also measure the fault detection rate according to the order of executing contexts.

B. *Results*

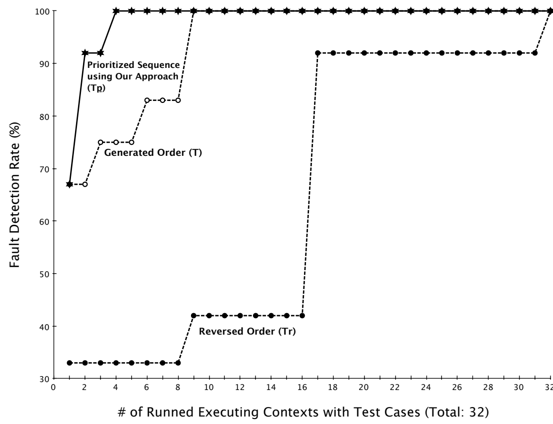
Number of detected bugs. We found 12 out of 38 bugs in Open Camera, and 14 out of 151 bugs in Subsonic (see Table IV). This results show that our testing is useful in detecting faults.

APFD measure. For Open Camera, # of contexts running with test cases (n) is 32, and # of faults (m) is 12. The APFD for three orders (i.e., T (generated order), T_r (reversed order), T_p (prioritized order using our approach)) are calculated as follow:
T: 0.92 = 1-(6+1+1+1+1+9+9+3+1+1+1+1)/384,
T_r: 0.62 = 1-(3+9+17+17+17+1+1+1+32+17+17+17)/384,
T_p: 0.97 = 1-(4+1+1+1+1+2+2+2+1+1+1+1)/384.
For Subsonic, # of contexts running with test cases (n) is 128, and # of faults (m) is 14. The APFD for three orders are calculated as follow:
T: 0.96 = 1-(2+1+1+1+1+2+1+1+6+6+1+1+1+1)/1536,
T_r: 0.92 = 1-(10 +1+1+1+1+10+1+1+5+5+1+1+1+1)/1536,
T_p: 0.98 = 1-(1+11+11+11+19+1+11+20+11+11+4+11+3+11)/1536.
In both projects, the APFDs for T_p represent the highest scores. Note that, in Subsonic, the number of generated executing contexts is large (i.e., 128) and many of the faults are detected by small number of the executing contexts, thus, the APFD measures are not much different in three orders.

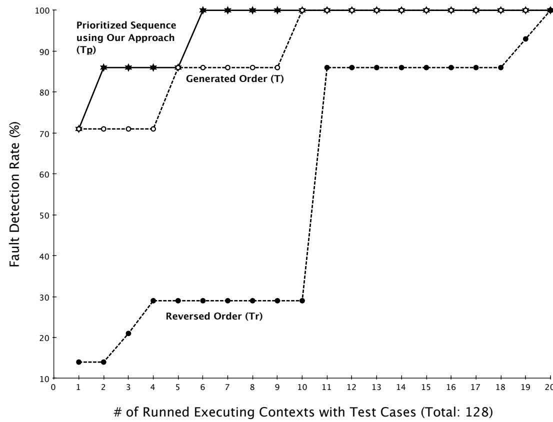
Fault detection rate. We present the graphs of fault detection rate for Open Camera and Subsonic in Fig. 2(a) and in Fig. 2(b), respectively. The graphs show the percentages of faults detected versus the fraction of the contexts used, for each sequence of comparators. For Open Camera, we (T_p) reached to 100% of fault detection rate after running four executing contexts, while the generated order (T) and the reversed order (T_r) required 9 and 32 executing contexts, respectively. For Subsonic, our prioritized sequence (T_p) reached to 100% of fault detection rate after running six executing contexts, while the generated order (T) and the reversed order (T_r) required 10 and 20 executing contexts, respectively. From the results, we can conclude that the order prioritized using our prioritization method results in the earliest detection of the faults.

VI. Conclusion and Future Work

In our paper, we provide a method for systematically generating various executing contexts from permissions to



(a) Fault detection rate for Open Camera.



(b) Fault detection rate for Subsonic.

Fig. 2: Fault detection rate graphs.

test Android applications. To generate the various contexts, the related resources and their possible states are identified from the permissions. Then, the various executing contexts are generated by permuting resource conditions, and the executing contexts are prioritized and selected. We applied our testing method to two open-source projects and showed the method is effective in fault detection.

For future work, we plan to consider more permissions and identify the relations between the resources and those permissions. We also plan to perform the more detailed experiment for showing the capability of using various contexts. Finally, we plan to devise the method of considering sequences in our contexts for simulating dynamically changing environment.

ACKNOWLEDGMENT

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education(NRF-2014R1A1A2054098). This research was supported by the MSIP(Ministry of Science, ICT and Future Planning), Korea, under the ITRC(Information Technology Research Center) support program (IITP-2015-H8501-15-1012) supervised by the IITP(Institute for Information & communications Technology Promotion).

REFERENCES

[1] J. Dehlinger and J. Dixon, "Mobile application software engineering: Challenges and research directions," in *Workshop on Mobile Software Engineering*, 2011.

[2] Peak Vision, http://www.peekvision.org/.

[3] M. Baldauf, S. Dustdar, and F. Rosenberg, "A survey on context-aware systems," *International Journal of Ad Hoc and Ubiquitous Computing*, vol. 2, no. 4, pp. 263–277, 2007.

[4] D. Amalfitano, A. R. Fasolino, P. Tramontana, and B. Robbins, "Testing android mobile applications: Challenges, strategies, and approaches." *Advances in Computers*, vol. 89, pp. 1–52, 2013.

[5] H. Muccini, A. Di Francesco, and P. Esposito, "Software testing of mobile applications: Challenges and future research directions," in *Automation of Software Test (AST), 2012 7th International Workshop on*. IEEE, 2012, pp. 29–35.

[6] D. Amalfitano, A. R. Fasolino, P. Tramontana, and N. Amatucci, "Considering context events in event-based testing of mobile applications," in *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 126–133.

[7] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles, "Towards a better understanding of context and context-awareness," in *Handheld and ubiquitous computing*. Springer, 1999, pp. 304–307.

[8] 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), 2010 IEEE 21st International Symposium on*. IEEE, 2010, pp. 249–258.

[9] System Permissions, http://developer.android.com/intl/ko/guide/topics/security/permissions.html.

[10] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and G. Imparato, "A toolset for gui testing of android applications," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 650–653.

[11] Open Camera, http://opencamera.sourceforge.net.

[12] Subsonic, http://subsonic.org/pages/apps.jsp#android.

[13] UI/Application Exerciser Monkey, http://developer.android.com/tools/help/monkey.html.

[14] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 258–261.

[15] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "A gui crawling-based technique for android mobile application testing," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*. IEEE, 2011, pp. 252–261.

[16] M. Wang, J. Yuan, H. Miao, and G. Tan, "A static analysis approach for automatic generating test cases for web applications," in *Computer Science and Software Engineering, 2008 International Conference on*, vol. 2. IEEE, 2008, pp. 751–754.

[17] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 435–445.

[18] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Permission evolution in the android ecosystem," in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 31–40.

[19] Menifest Permissions, http://developer.android.com/reference/android/Manifest.permission.html.

[20] R. Johnson, Z. Wang, C. Gagnon, and A. Stavrou, "Analysis of android applications' permissions," in *Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on*. IEEE, 2012, pp. 45–46.

[21] Open Camera Bug Issues, http://sourceforge.net/p/opencamera/tickets/.

[22] Subsonic Bug Issues, http://sourceforge.net/p/subsonic/bugs.

[23] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *Software Engineering, IEEE Transactions on*, vol. 27, no. 10, pp. 929–948, 2001.