JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 38, 1149-1169 (2022)
DOI: 10.6688/JISE.202211_38(6).0004

An Operation-Injection Approach to Detect Runtime Permission Crashes of Android Apps

CHIEN-HUNG LIU[1] AND SHU-LING CHEN[2,+]
[1]*Department of Computer Science and Information Engineering*
National Taipei University of Technology
Taipei, 106 Taiwan
[2]*Department of Industrial Management and Information*
Southern Taiwan University of Science and Technology
Tainan, 700 Taiwan
E-mail: cliu@ntut.edu.tw; slchen@stust.edu.tw

Starting from Android API 23, Android apps need to request appropriate runtime permissions before accessing restrict data or performing restrict actions, such as reading files or taking photos. Further, users can revoke the permissions that were previously granted to an app in system settings later or even during runtime of the app by keeping the app in background, going to system settings to disable the permissions, and returning back to the app. This can cause an app to crash if the app doesn't handle the runtime permissions carefully. To automatically detect the crashes related to runtime permissions, this paper proposes an approach in which a crawler is first used to explore and detect permission crashes of Android apps systematically. During the exploration, a state model is also produced. Based on the model, test paths related to runtime permissions are generated. These test paths are further injected with operations to revoke the already granted permissions and executed using a test runner directly to detect the crashes that can occur if users disable the granted permissions manually. The experimental results show that the proposed approach can detect runtime permission crashes effectively.

Keywords: Android DevOps, crash detection, runtime permissions, Android crawler, operation injection

1. INTRODUCTION

Recently, DevOps [1] for Android apps is gaining popularity since apps are often constantly updated due to frequent release of new smartphones, OS updates, feature enhancements, and quality improvements. Particularly, to react faster to user feedback, Android DevOps practices [2] become indispensable for streamlining the app development process and bringing together different stakeholders so that the development and delivery time of Android apps can be largely reduced. Among various practices of Android DevOps, automated app testing is considered essential since different app versions needed to be tested rapidly on numerous smartphone devices before an updated app can be released.

There are many automated testing tools that can be used in the Android DevOps environment to assure the quality of apps from different perspectives, such as functionality, compatibility, and security. In particular, in the DevOps environment, an updated Android app should be continuously tested to see if the app behaves as expected and no runtime crashes will occur. To ensure that Android apps function properly, many approaches and tools have been proposed to detect app runtime crashes automatically. However, to detect

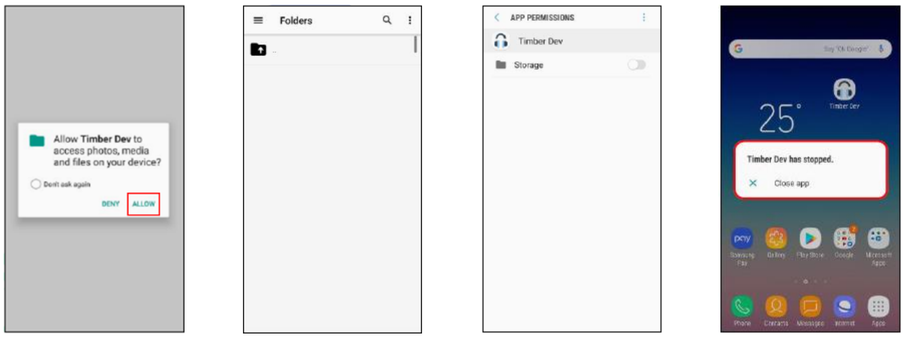
Received December 30, 2021; revised February 9, 2022; accepted March 9, 2022.
Communicated by Shin-Jie Lee.
[+] Corresponding author.

all possible crashes of Android apps thoroughly is challenging, since the causes of the crashes can be very wide and diverse, such as incorrect callback implementations or lack of considering Android versions. Thus, more efforts are still needed to tackle this problem.

Particularly, starting with Android API 23 (*i.e.*, Android version 6), Google provides a new runtime permission model to replace the old install-time permission system to further protect user privacy [3]. Unlike the install-time permissions which are automatically granted when an app is installed, in the new model users will be prompted to accept or deny individual permission requests at runtime when an app attempts to access restricted data like user's location and contact or perform restricted actions like taking photos and recording audio. To access the resources protected by the runtime permissions, also known as *dangerous permissions*, developers must declare the permissions and provide the corresponding implementations that follow a predefined workflow [4] (to be detailed in Section 2.1) for obtaining the permissions and handling user's responses. Incorrect implementations for the workflow of each individual runtime permission required by an app can cause the app to crash.

Moreover, users can revoke the permissions that were previously granted to an app in system settings later or even during the runtime of the app by keeping the app in background, going to system settings to disable the individual permissions granted to the app, and returning back to the app. This indicates that even though a resource protected by runtime permission can be accessed now, there is no guarantee that the resource can be successfully accessed next time. Thus, a new class of bugs can be introduced by the runtime permission model and dissatisfy users [5] if the implementations of an app do not always request and obtain the permissions before the app accessing any restricted data or performing restricted actions.

Fig. 1 shows a motivation example for an app that requires to access the photos, media, and files on the devices. To access these restricted resources, the app first requests and obtains the storage permission from the user as shown in Fig. 1 (a). If the requested permission is allowed by the user, the app then can successfully access the resource files in the folder as shown in Fig. 1 (b). Later, the user can go to app setting and disable the storage permission that was previously granted to this app as shown in Fig. 1 (c). If the implementation of the app does not always check and obtain the storage permission before accessing the files in the folder, then a runtime permission crash can happen as shown in Fig. 1 (d) when the app tries to access the files again.


Allow Timber Dev to access photos, media and files on your device?

Timber Dev has stopped.

(a) Allowing permission. (b) Accessing files. (c) Disabling permission. (d) App crash.

Fig. 1. An example of permission crash happened when revoking a previously granted permission.

To automatically detect the crashes related to runtime permissions, this paper proposes an approach based on app crawling and operation injection. Particularly, the proposed approach consists of two phases. The first phase is to systematically explore Android apps using a crawler to detect possible crashes including those crashes related to runtime permissions. Specifically, during this phase, the behavior of the app under test (AUT) is systematically exercised including the behavior related to deny and allow the runtime permission requests. A state model of the AUT is also created by the crawler. In the second phase, the exploring sequences (*i.e.*, test paths) related to runtime permissions are generated from the state model. These test paths are further injected with operations to revoke the already granted permissions and executed using a test runner directly to detect the crashes that can occur if users disable the granted permissions manually.

To support the proposed approach, we have first extended an Android crawler called ACE [6] for properly exploring the app behavior related to runtime permissions and detecting possible runtime permission crashes. The state model generated by the extended ACE is then used by the tool called AAD (Android Anomaly Detector) to further detect other runtime permission crashes using revoke-operation injections. The AAD used in the proposed approach is an extension from our previous work [7]. Specifically, AAD implements a test runner to execute the operation-injected test paths directly without the need to generate test scripts. Moreover, AAD can also run multiple test cases associated with a single test path together. Thus, it can reduce the overall test case execution and, hence, improve the detection efficiency as compared with the tool used in the earlier work of this paper [8]. To evaluate the approach, several experiments were conducted. The experimental results show that the proposed approach can effectively detect runtime permission crashes of Android apps. Further, it requires much less execution time than the earlier work.

The rest of the paper is organized as follows. Section 2 briefly reviews background and related work. Section 3 presents the proposed approach for detecting crashes related to runtime permissions. Section 4 describes and discusses the experimental results. The conclusion remarks and future work are given in Section 5.

2. BACKGROUND AND RELATED WORK

2.1 Background of Android Runtime Permissions

To protect apps and Android system from other or malicious apps, Android runs apps in a sandbox. If an app needs to access the resources outside the sandbox, the app must request and obtain the assess permissions corresponding to the resources. Beginning with Android 6.0 (API level 23), Google introduced a new permissions model that let apps request needed permissions from the user at runtime, rather than prior to installation. Further, users can revoke the already granted permissions from any app at any time on the device with API 23 or higher, even if the app targets a lower API level. Thus, even if an app can access restricted data or perform restricted actions now, it cannot assume that it still has the access permissions next time.

If an app needs a dangerous permission for accessing restricted data or performing restricted actions, the developer of the app must programmatically check whether the app has the permission every time by following a predefined permission request flow. If, however, the app does not properly follow the permission request flow to handle the accesses to restricted data or restricted actions, the app can crash or function unexpectedly.

Fig. 2 shows the Android runtime permission request flow [9]. Basically, the app first needs to check the Android platform version in which the app is running. If the platform is Android 6.0 or higher, it then checks whether the required runtime permission has been granted by user. If so, the app can access the restricted data or restricted actions. If not, the app may show a permission rationale dialog to explain why the app needs this particular runtime permission. The app then requests the runtime permission and handles the response depending on whether the user grants the requested permission or not.

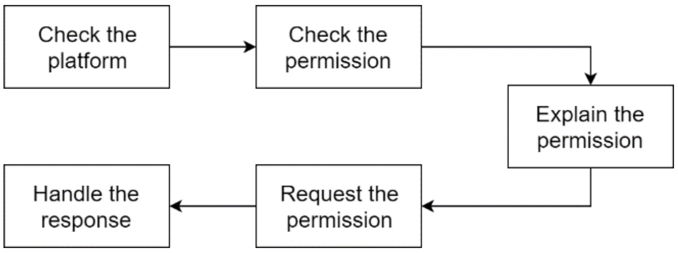


Fig. 2. The workflow for requesting and handling Android runtime permissions.

Fig. 3 shows a sample code that implements the aforementioned permission request flow for accessing the contacts on the smartphone. Suppose that the Android version for smartphone is API 23 or higher. To access the contact information, the app first uses the ContextCompat.checkSelfPermission() function to check whether the app has the permission to read the contact information. If the app has the permission, it then handles the request. If, however, the app does not have the permission (*i.e.*, READ_CONTACTS permission is not granted), it then uses the AcitivityCompat.shouldShowRequestPermission-Rationale() function to determine whether or not to show an explanation why the app needs this permission. The function will return true if the app has requested this permission before and the user denied the request. If the users denied the permission request before and checked the "Don't ask again" option, the function will return false. Finally, the app uses the ActivityCompat.requestPermissoins() function to request the permission from the user for accessing the contact information.

To handle user's response of the permission request, the app needs to implement the onRequestPermissionsResult() function as shown in Fig. 4. Based on the permission request code (*i.e.*, requestCode), the function selects the corresponding handler to process the request. For example, the requestCode for reading contacts is REQUEST_CODE_ READ_CONTACTS. Once the user responds to the permission request, the app then checks whether the user grants or rejects the request, and performs the corresponding actions based on user's response. If an app does not properly implement the permission workflow illustrated in Figs. 3 and 4, the app can crash whenever accessing restricted data or performing restricted actions.

```
public void showContacts(View view) {
  // Check for the requested permission
  if (ContextCompat.checkSelfPermission(MainActivity.this,
      Manifest.permission.READ_CONTACTS) != PackageManager.PERMISSION_GRANTED) {

      // Permission is not granted.   Should we show an explanation?
      if (ActivityCompat.shouldShowRequestPermissionRationale(MainActivity.this,
          Manifest.permission.READ_CONTACTS)) {
          // Show an explanation to the user…
      }
      //Request the permission
      ActivityCompat.requestPermissions(MainActivity.this,
            new String[]{Manifest.permission.READ_CONTACTS}, REQUEST_CODE_READ_CONTACTS);
  }
  else {
         // Permission has already been granted. Handle the request…
  }
```

Fig. 3. An example of sample code for checking and requesting a runtime permission.

```
@Override
public void onRequestPermissionsResult(int requestCode, String[] permissions, int[] grantResults) {
  switch (requestCode) {
    case REQUEST_CODE_READ_CONTACTS: {
      if (grantResults.length>0 && grantResults[0]==PackageManager.PERMISSION_GRANTED) {
          // Permission was granted! Do the contacts-related tasks.
      }
      else {
        // Permission denied! Disable the functionality that depends on this permission.
      }
      return;
    }
    // other 'case' lines to check for other permissions this app might request...
  }
}
```

Fig. 4. An example of sample code for handling the response of a runtime permission request.

2.2 Related Work

To assure the quality of Android apps, many approaches and tools have been proposed to detect runtime crashes of Android apps automatically. Particularly, app runtime crashes can be triggered by different types of errors, unhandled exceptions, or methods such as combinations of GUI events, rotating devices, or disconnecting network. In addition to permission crashes, detection of other app crashes is also important as well for ensuring the quality of Android apps. Thus, many existing tools (*e.g.*, Android Monkey [10]) consider only the detection of typical app crashes through exploring combinations of GUI events. Although these tools may also reveal some crashes that happen to relate to permissions, they did not specifically address the detection of runtime permission crashes, especially the crashes caused by revoking previously granted permissions. The followings briefly review existing studies for detecting app runtime crashes, including permission crashes.

Chen *et al.* [7] propose a method and tool called AAD (Android Anomaly Detector) that can detect anomalies in Android apps. Specifically, AAD can generate test cases automatically from the state model produced by an Android crawler, called ACE [6]. The test cases systematically traverse each state of the app and perform the four kinds of operations at the same time, including rotating back and forth, switching Wi-Fi connection on/off, exiting and re-entering app, and entering random input data to test whether the app behave normally. This work extends AAD to detect runtime permission crashes by injecting operations into test paths generated from the state model to revoke the already granted permissions and running the test paths to see if any crashes can happen.

Adamsen *et al.* [11] proposed a crash detection tool called THOR for Android apps. Specifically, the tool can inject system events after the execution of each event in the test script to detect app crashes caused by environment changes or rotations of a smartphone. However, the tool needs to write assertions manually in the test script developed by testers in order to check whether app errors occur. Further, although the tool can detect the crashes caused by environment changes or rotations, it does not support the detection of runtime permission crashes.

Hu *et al.* [12] presented an app testing tool called AppDoctor. The tool can inject system or user actions to test if an Android app handles the changes of activity lifecycle correctly. Particularly, it uses approximate execution by directly invoking event handler to speed up the testing. Moreover, the tool can automatically verify the exposed bugs and remove most false positives. To further speed up bug diagnosis, an action slicing technique is used to reduce the lengths of action traces and simplify bug diagnosis. However, the tool mainly focuses on app crashes related to activity lifecycle and does not support runtime permission crash detection.

Moran *et al.* [13, 14] propose a testing tool for Android apps called CrashScope. The tool can systematically generate test inputs, explore, and detect whether an app crashes with the information obtained from static and dynamic analysis, such as contextual API calls. To detect crashes more effectively, CrashScope supports two text input generation and GUI traversing strategies. Particularly, the tool can automatically generate detailed crash reports and replay test scripts that are very useful for developers to reproduce the crashes. Although CrashScope can detect crashes caused by different factors, such as inputs, device rotation, and disconnected network, it does not yet support runtime permission crash detection.

Mao *et al.* [15] proposed a crash detection tool for Android apps called Sapienz based on search-based testing technique. In particular, the tool can automatically explore and optimize test sequences and minimize the length of test sequences while maximizing test coverage and fault revelation. Specifically, the tool combines random fuzzing, systematic and search-based exploration, and string seeding to generate test inputs and guide the exploration of the app under test. The experimental results indicate that Sapienz has better performance than other fuzzing test tools. However, the tool does not support the runtime permission crash detection.

Zhang *et al.* [16] propose a crash detection tool for Android App called CrashFuzzer. CrashFuzzer is mainly used to test app crashes due to poor input validation. Specifically, the tool combines static analysis and semi-random technique to generate test inputs for detecting crashes caused by improper input data processing. It identifies the API methods related to input data processing, performs exception handling analysis, generate input test data, and then injects the inputs into the app under test to check whether the app will crash. CrashFuzzer can also generate structured trace information of detected crashes to assist in the debugging of the app.

Su *et al.* [17] proposed an approach and a testing tool called Stoat (STOchastic model App Tester) for Android apps. The tool is divided into two execution phases. First, the app is explored to construct a GUI model (in the form of stochastic finite state machine). It then iteratively perturbs the probability values of the model transitions and generates test suites from the mutated stochastic model. The test suites are injected with system events to uncover possible errors or exceptions in an app. The proposed tool can detect app crashes caused by injected system events, and may detect the crashes caused by lack of requested permissions. However, it does not support the detection of app crashes caused by disabling the already granted permissions.

Cao *et al.* [18] proposed an Android testing tool called Xdroid. The tool can monitor and inject resource dependencies that an app requires, such as contents and permissions, for testing apps. In particular, to resolve dependencies, it makes use of the notification and injection mechanisms. If external resources or permissions are required during the testing, previous test data will be reused or the user will be notified to give the corresponding resources or permissions. To drive the test process, a built-in test tool based on Monkey is used. Although Xdroid can be used to detect crashes related to permissions, it does not deal with the crashes caused by revoking granted permissions.

Fang *et al.* [19] proposed a tool called revDroid to detect potential side effects of permission revocation for Android apps. Basically, the tool performs static analysis on app source code based on Soot [20] and FlowDroid [21]. It can analyze whether there is any code to check if the app has the permission before calling corresponding APIs, or whether the code will handle SecurityException. Unlike revDroid, the proposed approach uses a dynamic approach to explore an AUT for generating a state model while detecting possible app crashes. It then injects operations into generated test paths to revoke previously granted permissions for detecting runtime permission crashes. Thus, the proposed approach can also detect other types of crashes in addition to permission crashes.

Sadeghi *et al.* [22] proposed a tool called PATDroid (Permission-Aware GUI Testing of Android) for efficiently testing an Android app under a variety of permission settings. Particularly, the tool uses a hybrid analysis method to examine apps and their test suites to identify the permission combinations that are relevant to execution of tests. The irrelevant permissions are then excluded during the test runs in order to reduce the testing effort. Unlike PATDroid that focuses on the effort reduction for detecting permission crashes, the proposed tool aims to explore apps and generate test cases to detect permission crashes.

Another attempt was the earlier work of this paper [8], which reported a runtime permission crash detection approach for Android apps using the extended ACE and PAD (Permission Anomaly Detector). Instead of using PAD, this paper extends the AAD to support permission crash detection with a test runner and a re-run mechanism to further improve the efficiency of permission crash detection. The extension includes an additional background information describing the typical implementations of requesting and handling runtime permissions in an Android app, additional related work, an enhancement of the original approach by using a test runner with a re-run mechanism to reduce the execution time of test cases, and a new evaluation of the enhanced tool, *i.e.*, AAD, with more subject apps and its comparisons with the original one.

Table 1 shows the comparisons of the related tools and the proposed approach, including whether the tool can support permission crash detection, whether the tool can detect app crashes other than permission crashes, and whether the tool considers the permission crashes caused by revoking previously granted permissions.

Table 1. The comparisons of related tools and the proposed approach.

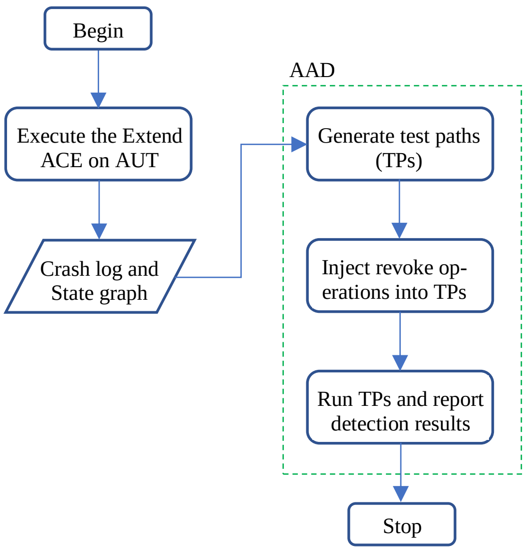
Testing Tool	Support permission crash detection	Detect crashes other than permission crashes	Detect permission crashes caused by permission revocation
THOR	×	✓	×
AppDoctor	×	✓	×
CrashScope	×	✓	×
Sapienz	×	✓	×
CrashFuzzer	×	✓	×
Stoat	✓	✓	×
Xdroid	✓	✓	×
revDroid	✓	×	✓
PATDroid	✓	✓	×
The proposed approach	✓	✓	✓

3. THE PROPOSED APPROACH

This section describes the proposed approach, including the overview of the approach, the extension of ACE for properly crawling permission dialogs and permission response behavior of apps, and the method of injecting permission-revoke operations to detect runtime permission crashes.

3.1 Overview of the Approach

The proposed detection process of runtime permission crashes is shown in Fig. 5. The first step uses the extended ACE [6] to explore the AUT, which generates a crash log and a GUI state graph (*i.e.*, state model) with all of the states found and the events (transitions) executed by the extended ACE. Each state of the GUI state graph is an instance of an Android activity and each event is a transition indicating that the event transfers a particular GUI state s_i to another GUI state s_j. The second step uses the AAD to generate a number of test paths (TP) from the GUI state graph according to ACE's exploring sequences related to runtime permissions. Each test path is a sequence of events that traverses the AUT from its root state to a leaf state. For each test path, the AAD then injects corresponding permission-revoke operations in-between each pair of events after the permission is granted or after the last event of the path. After that the AAD executes the test paths directly using a test runner to detect and report the crashes revealed by the injected operations.


Begin
AAD
Execute the Extend ACE on AUT
Generate test paths (TPs)
Crash log and State graph
Inject revoke operations into TPs
Run TPs and report detection results
Stop
Fig. 5. The proposed detection process of runtime permission crashes.

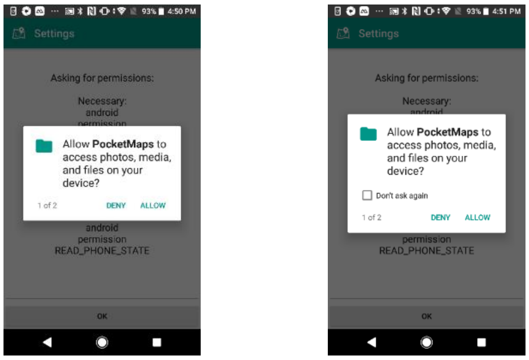
3.2 Extend ACE to Detect Runtime Permission Crashes

To enable ACE to correctly explore app behavior related to runtime permissions, ACE is extended so that (1) both events for denying and granting permission requests can be exercised; and (2) the response behavior of app for a granted permission can be explored properly.

Note that depending whether the permission is requested first time or not, Android system will display different permission dialogs. Fig. 6 (a) shows an example of a permission dialog, hereafter called AskPermission, that is displayed when a permission is requested first time by an app. If the user clicks the "Allow" option to grant the permission, this dialog won't appear next time. If, however, the user denies the request, the next time when the app requests the same permission again, the dialog shown in Fig. 6 (b), hereafter called AskPermissionAgain, will be displayed. If the user checks "Don't ask again" to deny the permission again, the app will no longer ask for this permission again until the user clears the app's setting.

To properly explore the app behavior of denying or allowing a permission, ACE is extended to crawl the permission dialog in a predefined order according to the type of the dialog. Particularly, the extended ACE will execute only the "Deny" event of the AskPermission dialog; otherwise, the event doesn't have a chance to be fired after the "Allow" event is executed. Similarly, the extended ACE will execute only the "Allow" event of the AskPermissionAgain dialog since the "Deny" event has been explored before in the AskPermission dialog.

Moreover, when the "Allow" option of a permission dialog is clicked, (*i.e.*, the permission request is granted), depending on the app implementation, the response behavior of an app can be (1) directly to access the restricted data or perform the restricted action to fulfill the request, or (2) indirectly to return to the original activity that triggers the permission request and let the user invoke the request again. In the latter case, when the request is invoked again, the app can access the requested resource or perform the requested action successfully since the requested permission has been granted. However, this can introduce a crawling problem because ACE may not be able to crawl the GUI of the original activity again if all the executable events in the original activity and the permission dialog have been explored. In such a case, the response behavior of the app for allowing the permission won't be explored correctly since ACE considers all the related GUI states and events being explored.


Asking for permissions:

Necessary:

Allow PocketMaps to access photos, media, and files on your device?

Don't ask again

READ_PHONE_STATE

(a) AskPermission dialog. (b) AskPermissionAgain dialog.
Fig. 6. The dialogs for AskPermission and AskPermissionAgain.

Figs. 7 and 8 respectively show two kinds of app response behavior that accesses the camera directly and accesses the storage media indirectly through the original activity when the "Allow" option of a permission dialog is clicked. Particularly, in Fig. 8, the GUI for the activity that triggers the storage permission dialog has only one button and was crawled when triggering the permission dialog. In this case, the GUI of this activity won't be explored by ACE again since all GUI events in the activity and permission dialog have been explored. To enable ACE to properly explore the response behavior of handling a granted permission for the case illustrated in Fig. 8, a hidden dummy event is added to the permission dialog so that ACE will return to the original activity for invoking the permission request one more time in order to execute the dummy event. Consequently, the response behavior of allowing the permission can be explored successfully.

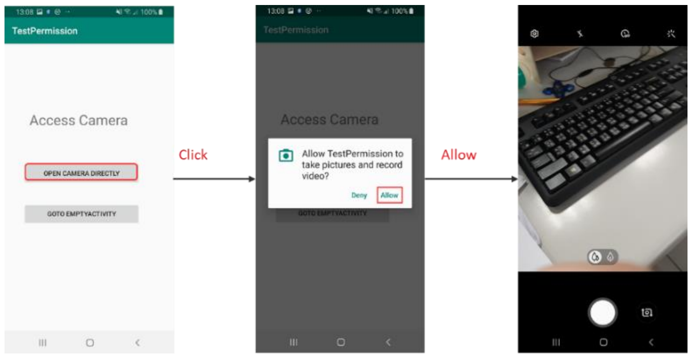

Allow TestPermission to take pictures and record video?

Fig. 7. An example of fulfilling a permission request directly when clicking the "Allow" option.

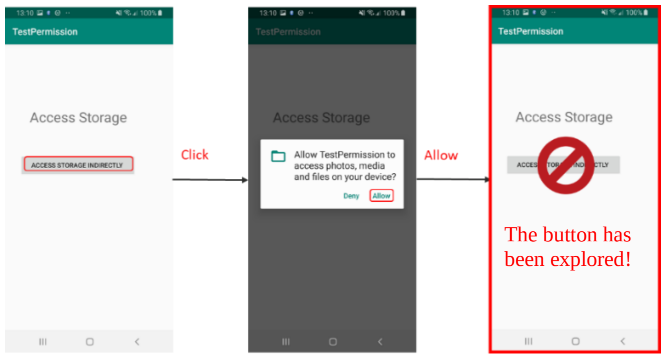

TestPermission
Access Storage
ACCESS STORAGE INDIRECTLY
Click
Allow TestPermission to access photos, media and files on your device?
Deny
Allow
Allow
The button has been explored!

Fig. 8. An example of fulfilling a permission request indirectly when clicking the "Allow" option.

3.3 Inject Revoke Operations to Detect Permission Crashes using AAD

The extended ACE can be used to explore the app behavior related to permission request and handling and, hence, can detect possible runtime permission crashes. However, it is still unable to detect the permission crashes that can occur when the user manually disables the granted permissions and then resumes the app. To detect such crashes and ensure that the apps request and obtain the permissions before accessing any restricted data or performing any restricted actions, the state graph generated by extended ACE is then used by AAD to create test paths (*i.e.*, exploring sequences) related to runtime permissions. Each generated test path is from the initial GUI state to a leaf GUI state of the state graph. The test paths are then injected a set of operations to revoke the already granted permissions.

To inject the permission-revoke operations, let Eq. (1) be a test path (*i.e.*, sequence of events) created from the state graph generated by the extended ACE. Suppose that event e_i is the event allowing a particular permission request (*i.e.*, e_{allow}). Since the user can revoke a permission at any time after it is granted and before the restricted resource is accessed, to detect possible runtime permission crashes caused by an event after e_{allow}, the proposed approach injects a permission-revoke operation, denoted as r, in-between each pair of the events after e_{allow} or after the last event of the test path as shown in Eqs. (2)-(5). For example, Eq. (2) is an injected test path that simulates a test scenario in which the permission is granted in e_{allow}, revoked by user right after e_{allow}, and then immediately followed by an event to access restricted data or perform restricted actions requiring the permission.

$$\langle e_1, \ldots, e_{i-1}, e_i, e_{i+1}, \ldots, e_n \rangle \tag{1}$$

$$\langle e_1, \ldots, e_{i-1}, e_{allow}, r, e_{i+1} \rangle \tag{2}$$

$$\langle e_1, \ldots, e_{i-1}, e_{allow}, e_{i+1}, r, e_{i+2} \rangle \tag{3}$$

$$\ldots$$

$$\langle e_1, \ldots, e_{i-1}, e_{allow}, e_{i+1}, \ldots, e_{n-1}, r, e_n \rangle \tag{4}$$

$$\langle e_1, \ldots, e_{i-1}, e_{allow}, e_{i+1}, \ldots, e_{n-1}, e_n, r \rangle \tag{5}$$

Note that a test path can have many injection points to inject revoke operations as illustrated in Eqs. (2)-(5). Thus, many test cases can be generated from a single test path. In our earlier work [8], multiple test cases are created from a single test path and each of them is converted into a test script to detect runtime permission crashes for Android apps automatically. Although such a method provides flexibility to developers to edit test scripts, it will generate a lot of test cases with small differences in the injection point of revoke operation and the length of the test sequence. To reduce the time for detecting permission crashes, this paper employs a test runner that can execute the operation-injected test paths on AUT directly without the need to convert test cases into test scripts. Moreover, to further improve the crash detection efficiency, test cases for a single test path are run together with a re-run mechanism to shorten the execution time of individual test cases.

Suppose that test cases $\langle e_1, \ldots, r, e_{n-1}\rangle$ and $\langle e_1, \ldots, e_{n-1}, r, e_n\rangle$ are two consecutive sequences derived from the same test path. We can observe that these two test cases have almost the same test sequences except for the second test case has one more event, *i.e.*, e_n in the end of its sequence than the first test case. Moreover, the revoke operation r is injected only once in the sequence before the event that could access restricted resource, *i.e.*, e_{n-1} or e_n. Note that the revoke operation can be injected at any time after the requested permission is granted and before the restricted resource is accessed. Therefore, for the second test case, if we change the injection point of the revoke operation r from before e_n to before e_{n-1}, we can obtain a test sequence $\langle e_1, \ldots, r, e_{n-1}, e_n\rangle$ which is equivalent to the second test case from the perspective of permission crash detection. Therefore, instead of executing both test cases separately, after executing r and e_{n-1} in the first test case, if no crash occurs (*i.e.*, e_{n-1} is not affected by the revoke operation r), we then can run the second test case simply by directly executing e_n right after the execution completion of the first test case. Thus, following this way, we can run multiple test cases derived from the same test path together instead of separately to further reduce the crash detection time.

The above scenario works if the injected revoke operation r does not cause AUT crash. However, if the AUT gets crashed due to the revoke operation, the crash state of AUT can interrupt the execution of the remaining test cases that are generated from the same test path. In order to resume and continue the execution of the remaining test cases, the AUT is restarted and a "re-run" mechanism is proposed and implemented in the test runner to re-inject the revoke operation and re-run the remaining test cases of the test path. In particular, assume that the revoke operation r is injected right before an event e_c and the execution of r and e_c causes AUT crash. In such a case, the "re-run" mechanism will (1) remove the revoke operation r injected before e_c to avoid this crash; (2) re-inject a new revoke operation into test path right after e_c to generate a new test sequence for the remaining test cases; and (3) start a new test run with the new test sequence. In this way, each AUT crash will trigger a new test run to remove the previously injected revoke operation, re-inject the revoke operation right after the event causing AUT crash, and re-run the test until the remaining test cases of the same test path are all executed.

To illustrate the idea of the re-run mechanism, let's consider an example test path TP $\langle e_1, \ldots, e_{allow}, e_c, \ldots, e_n\rangle$ shown in Fig. 9. Suppose that TP is initially injected with a revoke operation after e_{allow}. Thus, TP becomes as $\langle e_1, \ldots, e_{allow}, r, e_c, \ldots, e_n\rangle$. Assume that the first test run of TP (*i.e.*, test run: 1) causes an app crash after executing r and e_c. At this time, the test runner will remove the previously injected r from TP and re-inject r into TP after e_c to generate a new test sequence $\langle e_1, \ldots, e_{allow}, e_c, r, \ldots, e_n\rangle$. Then, the test runner starts the next test run (*i.e.*, test run: 2) with the new test sequence for the remaining test cases to detect other app crashes. The re-run process will repeat if any app crash is detected again until all the test cases associated with the same test path are executed.

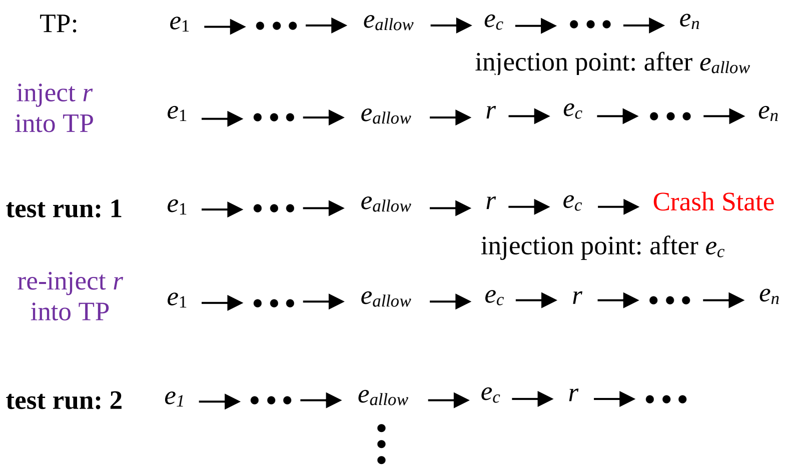

TP:
e_1 → ••• → e_{allow} → e_c → ••• → e_n
injection point: after e_{allow}
inject r into TP
e_1 → ••• → e_{allow} → r → e_c → ••• → e_n
test run: 1
e_1 → ••• → e_{allow} → r → e_c → Crash State
injection point: after e_c
re-inject r into TP
e_1 → ••• → e_{allow} → e_c → r → ••• → e_n
test run: 2
e_1 → ••• → e_{allow} → e_c → r → •••

Fig. 9. The re-run mechanism if a crash occurs when executing a test path.

Fig. 10 shows the algorithm of the test runner to inject permission revoke operation and run the test path directly for detecting runtime permission crashes. Basically, the algorithm takes a test path as input and outputs a crash report related to the test path. The function findFirstInjectionPoint() in Line 2 will return the event that grants a permission on the test path. The injectRevokeOperationIntoTestPath() function in Line 3 will inject a revoke operation into the test path right after a target event and generate a test sequence. The findFirstEvent() in Line 4 will find the first event of the test sequence to be executed. The while-loop in Lines 5-16 will execute the event in the test sequence one by one. If the execution of an event causes an app crash, find the injection point after the event (Line 9), re-inject a revoke operation into the test path to generate a new test sequence (Line 10), and record the crash in the report (Line 11). Then, find the first event of the new test sequence (Line 12) and re-run the test with the sequence for the remaining test cases of the test path (Line 13). Finally, Line 17 returns the crash report when all the events in the test path are executed completely.

Note that the permission-revoke operations being injected into a test path are corresponding to the types of granted permissions. The granted permission types, such as camera or file access, are logged by the extended ACE during the exploration and are recorded in the exploring sequences. Such information is provided to AAD along with the test paths in order to inject the proper revoke operations corresponding to the granted permission types into the test sequences. The detail to record the permission types is omitted here for simplicity.

Algorithm: PermissionCrashTestRunner

Input: TestPath *TP*
Output: CrashReport *report*

```
begin
  e ← findFirstInjectionPoint(TP)    // find the injection point to inject revoke operation
  itp ← injectRevokeOperationIntoTestPath(TP, e) // inject a revoke operation after e
  e ← itp.findFirstEvent()                              // find the first event in itp
  while e ≠ null
      execute e
      s ← current state
      if s is a crash state then       // re-inject a revoke operation after e and rerun TP
          re ← findInjectionPointAfterEvent(TP, e) // find next event after e to re-inject
          itp ← injectRevokeOperationIntoTestPath(TP, re)
          report ← record e and s
          e ←itp.findFirstEvent()
          continue
      endif
      e ← itp.FindNextEvent(e)    // find the next event after e in itp
  end while
  return report
end
```

Fig. 10. The algorithm of test runner for a given test path.

4. EVALUATION

We conduct experiments to study whether the extended ACE and the AAD are useful as well as whether the test runner approach can reduce the execution time of test cases as compared to the tool PAD used in our earlier work. The following three research questions are addressed:

RQ1 Can the extended ACE correctly explore the app behavior related to runtime permissions?
RQ2 Can the extended ACE and AAD effectively detect runtime permission crashes of an Android app from different perspectives?
RQ3 Can the test runner of AAD reduce the execution time of test cases as compared to PAD?

We select 13 apps, taken from F-Droid Market [1], as the subjects of our evaluation. These apps are selected because they use runtime permissions and some of them have been studied in the related work. The details of the subject apps, including the name of the app, line of code, the numbers of class, method, and activity, are given in Table 2. The hardware/software equipment used in the experiments is shown in Table 3. Two experiments were conducted to address the above research questions. Each experiment is described in the following subsections.

Table 2. The subject apps (AUT).

App	Line of Code	Class	Method	Activity
AnyMemo	28,178	190	6,427	30
Budget Watch	4,398	43	560	10
Catima	6,239	41	596	11
OSMBugs	4,127	55	2,450	3
ParkenDD	2,177	18	356	5
SAF Media Scanner	6,067	19	494	2
Smoke	27,458	45	1,668	7
Timber	20,379	159	3,568	7
Trigger	5,773	43	797	10
Rental Calc	4,124	24	428	12
PocketMaps	10,830	68	1,394	10
Etar	67,051	248	5,597	14
Open Note Scanner	3,587	23	654	4

Table 3. The experimental equipment.

Hardware/Software	Specification/Version
CPU	Intel Core i7-6700U 3.40 GHz
Memory	16GB LPDDR3 2133MHz
Disk	SSD 256GB + HDD 1TB
OS	Window10
Appium client/server	Version 4.1.2/1.20.1
JDK	Version 1.8.0
Android Studio	Version 4.1.1
Samsung A9	Android ver. 9.0.0

4.1 Experiment 1

The first experiment addresses RQ1. We use the extended ACE to explore the behavior of subject apps related to runtime permissions. The NFS algorithm [6] is used in ACE for minimizing the number of app restart and the timeout limit is set to 3 hours for giving ACE enough time to fully explore most apps. The results are shown in Table 4, including the statement coverage (S.C.) and branch coverage (B.C.), the execution time, the number of fired events, the number of explored states, the number of permission types requested by the apps, and the number of permission types granted to the apps. The results show that the original ACE is unable to explore eight out of thirteen apps, including AnyMemo, OSMBugs, SAF Media Scanner, *etc.*, since these apps request runtime permissions immediately when launching the apps. Thus, the code coverage of these apps is 0% for the original ACE. Note that, for the two apps Etar and Open Note Scanner, although they can be explored by the extended ACE successfully, we are unable to instrument these two apps on our smartphones running Android version 9. As a result, their coverage is unavailable.

Moreover, from the number of permission types granted to the apps, we can observe that the original ACE is unable to crawl the behavior of the subject apps for granting permissions and handling responses. The extended ACE, on the other hand, can successfully crawl the permission dialog and explore the behavior of denying or allowing a permission request for all subject apps. This can also be confirmed by the increases of code coverage, execution time, as well as the numbers of the fired events and explored states.

Table 4. The crawling results of extended ACE.

App	ACE version	S.C. / B.C.	Time (mm:ss)	Fired events	States	Num of perm. types requested	Num of perm. types granted
AnyMemo	extend	52% / 36%	181:36	2004	163	1	1
	original	0% / 0%	–	–	–		0
Budget Watch	extend	52% / 29%	141:29	1659	89	2	2
	original	29% / 13%	56:01	549	34		0
Catima	extend	22% / 13%	17:59	228	23	2	2
	original	16% / 8%	26:20	229	16		0
OSMBugs	extend	49% / 28%	15:15	227	26	1	1
	original	0% / 0%	–	–	–		0
ParkenDD	extend	14% / 10%	10:06	99	14	1	1
	original	14% / 9%	26:04	203	15		0
SAF Media Scanner	extend	6% / 3%	120:04	1322	48	1	1
	original	0% / 0%	–	–	–		0
Smoke	extend	7% / 3%	26:25	201	14	1	1
	original	0% / 0%	–	–	–		0
Timber	extend	37% / 22%	60:18	603	89	1	1
	original	0% / 0%	–	–	–		0
Trigger	extend	24% / 17%	74:48	759	36	2	2
	original	24% / 15%	50:28	454	28		0
Rental Calc	extend	79% / 52%	181:28	2006	82	1	1
	original	12% / 4%	12:41	108	11		0
PocketMaps	extend	3% / 2%	21:26	298	37	2	2
	original	0% / 0%	1:54	4	3		0
Etar	extend	unavailable	166:13	1702	117	3	3
	original	0% / 0%	–	–	–		0
Open Note Scanner	extend	unavailable	16:54	108	19	2	2
	original	0% / 0%	–	–	–		0

Overall, the answer to RQ1 is "yes, the extended ACE can correctly crawl the apps that use runtime permissions." Further, the extended ACE can obtain higher coverage than the original one.

4.2 Experiment 2

The second experiment addresses RQ2 and RQ3. We first use the extended ACE to explore the subject apps, generate state graphs, and discover runtime crashes related to permissions. Based on the generated state graphs, the AAD is then used to create test cases to detect runtime permission crashes that can occur when the user disables the already granted permissions. Again, the NFS algorithm was used in ACE and timeout limit is set to 3 hours. Tables 5 and 6 show the experimental results of the extended ACE and AAD, respectively. The results of Table 5 show that for the extended ACE, on average, the number of explored sequences is 73.5, the number of fired events is 862.5, the execution time of app exploration is 79.2 minutes, and the number of detected permission crashes is 0.2. In particular, the results show that the extended ACE indeed discovers runtime permission crashes for apps AnyMemo, ParkenDD, and Timber during the exploration.

Table 5. The detection results of the extended ACE.

App	Num of explored sequences	Num of events fired	Total execution time (min)	Num of permission crashes detected
AnyMemo	254	2,004	181	1
Budget Watch	76	1,659	141	0
Catima	21	228	18	0
OSMBugs	13	227	15	0
ParkenDD	18	99	10	1
SAF Media Scanner	153	1,322	120	0
Smoke	70	201	26	0
Timber	55	603	60	1
Trigger	91	756	74	0
Rental Calc	27	2,006	181	0
PocketMaps	27	298	21	0
Etar	133	1,702	166	0
Open Note Scanner	18	108	16	0
Average	**73.5**	**862.5**	**79.2**	**0.2**

Table 6. The detection results of AAD.

App	Num of created test paths	Num of injected operations	Total execution time (min)	Num of total perm. crashes detected	Num of unique perm. crashes detected
AnyMemo	1,314	1,208	1,147	552	1
Budget Watch	184	215	122	0	0
Catima	41	70	60	0	0
OSMBugs	112	100	96	0	0
ParkenDD	23	21	17	0	0
SAF Media Scanner	593	589	588	0	0
Smoke	252	252	174	0	0
Timber	170	129	154	111	5
Trigger	379	547	558	232	3
Rental Calc	214	213	236	99	1
PocketMaps	266	267	320	90	1
Etar	996	1,521	1,188	134	10
Open Note Scanner	93	18	95	4	1
Average	**356.7**	**396.2**	**365.8**	**94.0**	**1.7**

In addition, the results of Table 6 show that the AAD is able to detect app permission crashes happened when the already granted permissions were revoked. The results indicate that for AAD, on average, the number of created test paths is 356.7, the number of injected revoke operations is 396.2, the execution time of the test cases is 365.8 minutes, the number of detected permission crashes is 94.0, and the number of unique permission crashes is 1.7. Moreover, from the results of Tables 5 and 6, we can also observe that the number of test cases (*i.e.*, test sequences) created by AAD is much larger than that of the sequences explored by the extended ACE. This is because that the revoke operations can be injected after any subsequent event of the permission granted event since the user could disable an already granted permission at any time. Consequently, the number of fired events and the total execution time of AAD can be significantly larger than those of extended ACE.

Note that, for the ParkenDD app, the extended ACE can reveal a runtime permission crash. However, AAD does not discover this crash because the crash happens to occur in the exploring path of denying the permission request which is not covered by AAD. Further, for the Timber app, AAD can detect more runtime permission crashes than the extended ACE. This suggests that the extended ACE and AAD can detect runtime permission crashes from different perspectives and together they can provide a more comprehensive detection of runtime permission crashes for Android apps.

Overall, the extended ACE can detect runtime permission crashes during app exploration though it is unable to discover those permission crashes happened when the user disables the already granted permissions. On the other hand, AAD was able to detect such permission crashes through revoke-operation injections. Thus, the answer to RQ2 is "yes, the extended ACE and AAD can effectively detect runtime permission crashes from different perspectives and together they can provide a more complete detection of runtime permission crashes."

To evaluate whether the proposed test runner approach can reduce the execution time of test cases and result in a better permission crash detection efficiency than the method used in PAD, we compare the results of 4 apps that were used in the experiments of our earlier work [8]. Table 7 shows the comparisons of the experimental results between PAD and AAD. From the results, we can observe that the number of created test sequences, the number of fired events, and the total execution time of AAD are much less than those of PAD. The percentage of time reduction can be very significant although it varies depending on the AUT.

Table 7. The comparisons of execution results between PAD and AAD.

App	Testing Tool	Num of test sequences created	Num of events fired	Total execution time (min)	Percentage of execution time reduction
Open Note Scanner	PAD	592	4,790	775	–
	AAD	93	503	95	87.7%
PocketMaps	PAD	749	4,078	646	–
	AAD	266	1,374	320	50.5%
Etar	PAD	2,066	51,366	4,553	–
	AAD	996	8,474	1,188	73.9%
Timber	PAD	2,525	32,781	4,797	–
	AAD	170	1257	154	96.8%

Overall, AAD requires much less time than PAD to run the test cases for detecting runtime permission crashes. Thus, the answer to RQ3 is "yes, AAD can significantly reduce the execution time of test cases as compared to PAD."

5. CONCLUSIONS AND FUTURE WORK

This paper proposed an approach and tool, called AAD, that can detect runtime permission crashes of Android apps. Especially, the approach and tool can detect the crashes that can occur by manually disabling the granted permissions and then resuming the apps.

In particular, the proposed approach first explores the AUT systematically to detect app crashes related to runtime permissions and generates a state graph. Based on the state graph, test paths are generated automatically. For each test path, the approach injects revoke operations in-between each pair of the events after the permission is granted. A test runner is designed to execute the test path directly and report the revealed crashes. Our evaluation results showed that the proposed approach indeed is useful and promising for detecting runtime permission crashes. The use of test runner can also improve the efficiency of crash detection significantly.

The current approach mainly focuses on the detection of runtime permission crashes. Thus, other types of crashes caused by device rotation or disconnected network are not considered. In the future, we expect to expand the approach to detect more types of app crashes and further improve the detection efficiency of AAD. We also plan to integrate AAD into a mobile DevOps platform to support continuous testing of Android apps.

ACKNOWLEDGMENT

This research was partially supported by the Ministry of Science and Technology, Taiwan, under contract numbers MOST 110-2221-E-027-004, which is gratefully acknowledged.

REFERENCES

1. R. Jabbari, N. B. Ali, K. Petersen, and B. Tanveer, "What is DevOps?: a systematic mapping study on definitions and practices," in *Proceedings of the Scientific Workshop Proceedings of XP2016*, 2016, pp. 1-11.
2. R. Tak and J. Modi, *Mobile DevOps: Deliver Continuous Integration and Deployment within Your Mobile Applications*, Packt Publishing, Birmingham, 2018.
3. Android 6.0 Changes, https://developer.android.com/about/versions/marshmallow/android-6.0-changes, 2021.
4. Permissions on Android, https://developer.android.com/guide/topics/permissions/overview, 2021.
5. G. L. Scoccia, S. Ruberto, I. Malavolta, M. Autili, and P. Inverardi, "An investigation into Android run-time permissions from the end users' perspective," in *Proceedings of IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems*, 2018, pp. 45-55.
6. C. H. Liu, W. K. Chen, and S. H. Ho, "NFS: an algorithm for avoiding restarts to improve the efficiency of crawling Android applications," in *Proceedings of the 42nd IEEE International Conference on Computers, Software, and Applications*, 2018, pp. 69-74.
7. W. K. Chen, C. H. Liu, and Z. Z. Li, "On detecting abnormal behavior in Android apps," in *Proceedings of International Joint Conference of TCSE, JASPIC and SEA*, 2019.
8. C. H. Liu, C. T. Liu, and H. H. Li, "Detecting permission crashes of Android apps using crawling and revoke operation injections," in *Proceedings of the 28th Asia-Pacific Software Engineering Conference Workshops*, 2021, pp. 47-51.

9. Request App Permissions, https://developer.android.com/training/permissions/requesting, 2021.
10. Android UI/Application Exerciser Monkey, https://developer.android.com/studio/test/other-testing-tools/monkey, 2022.
11. C. Q. Adamsen, G. Mezzetti, and A. Møller, "Systematic execution of Android test suites in adverse conditions," in *Proceedings of International Symposium on Software Testing and Analysis*, 2015, pp. 83-93.
12. G. Hu, X. Yuan, Y. Tang, and J. Yang, "Efficiently, effectively detecting mobile app bugs with AppDoctor," in *Proceedings of the 9th European Conference on Computer Systems*, 2014, pp. 1-15.
13. K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically discovering, reporting and reproducing Android application crashes," in *Proceedings of IEEE International Conference on Software Testing, Verification and Validation*, 2016, pp. 33-44.
14. K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "CrashScope: a practical tool for automated testing of Android applications," in *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering*, 2017, pp. 15-18.
15. K. Mao, M. Harman, and Y. Jia, "Sapienz: multi-objective automated testing for Android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 94-105.
16. A. Zhang, Y. He, and Y. Jiang, "CrashFuzzer: detecting input processing related crash bugs in Android applications," in *Proceedings of the 35th International Performance Computing and Communications Conference*, 2016, pp. 1-8.
17. T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based GUI testing of Android apps," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 245-256.
18. C. Cao, C. Meng, H. Ge, P. Yu, and X. Ma, "Xdroid: testing Android apps with dependency injection," in *Proceedings of IEEE 41st Annual Computer Software and Applications Conference*, 2017, pp. 214-223.
19. Z. Fang, Z. Qian, and H. Chen, "revDroid: code analysis of the side effects after dynamic permission revocation of Android apps," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016, pp. 747-758.
20. R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot – a Java bytecode optimization framework," in *Proceedings of Conference of the Centre for Advanced Studies on Collaborative Research*, 1999, pp. 214-224.
21. S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel, "FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 259-269.
22. A. Sadeghi, R. Jabbarvand, and S. Malek, "PATDroid: permission-aware GUI testing of Android," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 220-232.
23. F-Droid, https://www.f-droid.org/, 2021.

Chien-Hung Liu (劉建宏) received his Ph.D. degree in Computer Science and Engineering from the University of Texas at Arlington in 2002. He is currently an Associate Professor of Computer Science and Information Engineering Department at National Taipei University of Technology, Taiwan. His research interests include software testing, software engineering, deep learning applications, and vocal detection.

Shu-Ling Chen (陳淑玲) received her Ph.D. degree in Industrial and Manufacturing Systems Engineering from the University of Texas at Arlington in 2002. She is currently an Assistant Professor of the Industrial Management and Information Department at Southern Taiwan University of Science and Technology. Her research interests include information management systems, e-business, supply chain, and software testing.