DeltaDroid: Dynamic Delivery Testing in Android

NEGAR GHORBANI, University of California, Irvine
REYHANEH JABBARVAND, University of Illinois Urbana-Champaign
NAVID SALEHNAMADI, JOSHUA GARCIA, and SAM MALEK, University of California, Irvine

Android is a highly fragmented platform with a diverse set of devices and users. To support the deployment of apps in such a heterogeneous setting, Android has introduced *dynamic delivery*—a new model of software deployment in which optional, device- or user-specific functionalities of an app, called *Dynamic Feature Modules (DFMs)*, can be installed, as needed, after the app's initial installation. This model of app deployment, however, has exacerbated the challenges of properly testing Android apps. In this article, we first describe the results of an extensive study in which we formalized a defect model representing the various conditions under which DFM installations may fail. We then present DeltaDroid—a tool aimed at assisting the developers with validating dynamic delivery behavior in their apps by augmenting their existing test suite. Our experimental evaluation using real-world apps corroborates DeltaDroid's ability to detect many crashes and unexpected behaviors that the existing automated testing tools cannot reveal.

CCS Concepts: • **Software and its engineering → Software testing and debugging;**

Additional Key Words and Phrases: Software testing, dynamic delivery, test augmentation, Android applications

ACM Reference format:
Negar Ghorbani, Reyhaneh Jabbarvand, Navid Salehnamadi, Joshua Garcia, and Sam Malek. 2023. DeltaDroid: Dynamic Delivery Testing in Android. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 84 (May 2023), 26 pages.
https://doi.org/10.1145/3563213

1 INTRODUCTION

The Android framework runs on diverse types of devices that vary both in terms of hardware properties, e.g., CPU architectures or hardware sensors, and software configuration, e.g., Android version or default language. Due to this variability, developers need to customize features or offer optional features for specific devices. More importantly, to ensure an app runs properly on different devices, developers need to include various device-specific resources and features in their apps. This strategy forces users to download larger apps with unnecessary code and resources specific to other devices. Alternatively, developers can maintain and publish multiple device-specific APKs

Authors' addresses: N. Ghorbani (corresponding author), N. Salehnamadi, J. Garcia, and S. Malek, Informatics Department, Donald Bren School of Information and Computer Sciences, University of California, Irvine, Interdisciplinary Science and Engineering Building, University of California, Irvine, Irvine, CA 92697; emails: negargh, nsalehna, joshug4, malek@uci.edu; R. Jabbarvand, Department of Computer Science, University of Illinois, 2106 Siebel Center, 201 N Goodwin Avenue, Urbana, IL 61801; email: reyhaneh@illinois.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1049-331X/2023/05-ART84 $15.00
https://doi.org/10.1145/3563213

for a single app, requiring them to spend additional effort to build, sign, upload, and manage several APKs.

To address these issues, Android has recently introduced a new publishing format, called *Android App Bundle* [10]. One of the main goals of app bundles is to enable *dynamic delivery* [24], i.e., enhance modularization so that developers can customize their apps based on user requirements and deliver optional features on a user's demand. To that end, each app bundle consists of several cohesive modules that together implement an app's full functionality. Specifically, the *base module* of the app bundle implements the core functionality of the app, *configuration modules* include device-specific resources, and ***Dynamic Feature Modules*** (**DFMs**) implement optional features. When users install an app for the first time, they install only the base module and a configuration module specific to their devices. DFMs can be downloaded later based on the user's demand.[1] To install a DFM, the app's base module should send an *installation request* to the Android app store operator through its runtime interface, i.e., the app store API. Once the app store operator accepts the request, the app downloads the requested DFM and installs it on the user's device. The lifecycle of an installation request is not always straightforward. In fact, such requests may *fail* due to several reasons and result in critical failures, e.g., crashes [25]. Android documentation has provided instances of best practices regarding dynamic delivery [27], suggesting that developers monitor the state of an installation request throughout its lifecycle and transparently communicate to the user if a failure happens. More specifically, if a DFM installation fails, Android best practice suggests that developers notify the user about the root cause of the failure and adjust the app's behavior.

As a result, to ensure an app's proper behavior in dynamic delivery, developers require tests that (1) reach the installation request in the code and (2) induce the contexts in which the installation request fails. The challenge here lies in the fact that such failures happen under peculiar conditions, e.g., network disconnection, missing the proper permission, or no support for the app store API on the requesting device. Since these conditions cannot be effectively produced through GUI actions, most existing Android testing techniques (e.g., [18, 44, 56]) that target GUI testing are not suitable for testing an app's dynamic delivery. While Android documentation provides a list of conditions that may cause an installation request to fail [24], the description of those conditions is both vague and incomplete, making developers struggle with dynamic delivery testing and understanding the conditions under which dynamic delivery fails [2–4, 7–9, 13–16].

To overcome these challenges, we propose DeltaDroid, an approach for **D**ynamic D**EL**ivery **T**esting of **An**Droid, i.e., a test-suite augmentation approach that leverages static and dynamic analysis techniques to (1) identify tests that reach DFMs' installation requests, and (2) modify the identified tests with a combination of GUI and system actions to generate new test cases that induce different conditions, which may cause a DFM installation request to fail. To identify proper actions, it relies on a novel defect model that formally describes the contexts in which failures occur. Such tests will help developers with the steps that lead to DFM installation failure, and they only need to specify application-specific assertions.

Given the fact that Dynamic Delivery has been widely used in recent Android apps [11] (over 40% of all app releases on Google Play [23]), validating the correctness of Dynamic Delivery is crucial. Our evaluation shows that while state-of-the-art [44] and state-of-the-practice [18] Android testing techniques, which mostly focus on GUI testing, are potentially able to generate tests that can initiate the installation of DFMs, they are not capable of creating the proper conditions to make an installation request fail. More specifically, we detected 44 critical dynamic delivery failures, i.e., crashes, across 25 subject apps. After reporting these failures to developers, 18 of them were verified to date.

[1]Developers define how and when users can download different dynamic features on devices running Android 5.0 (API level 21) or higher.

Fig. 1. Installation of a new DFM on K9Mail.

This article makes the following contributions:

- Construction of a defect model representing Android dynamic delivery failure through the formalization of failure conditions.
- A hybrid static and dynamic analysis technique for augmenting existing test suites with tests that can validate Android dynamic delivery. An implementation of the proposed technique is publicly available [20].
- An extensive empirical evaluation of real-world Android apps demonstrating DeltaDroid's effectiveness in test augmentation.

The remainder of this article is organized as follows. Section 2 illustrates a motivating example. Section 3 introduces the formal specifications of the defect model. Section 4 describes the details of DeltaDroid and its implementation. Section 5 presents the experimental evaluation of the research. The article concludes with a discussion of the related research and avenues of future work.

2 ILLUSTRATIVE EXAMPLE

As an illustrative example, we use the Android app K9Mail [12]. We describe the functionality of K9Mail, a defect in the app associated with dynamic delivery, and how a test can reveal the defect.

App: K9Mail is an e-mail client app whose main functionalities are viewing and sending e-mails, managing multiple e-mail accounts, searching, and so on. In addition to these core functionalities, K9Mail offers an optional feature for encrypting e-mails to provide an additional level of security. Since most users do not use this feature, it is encapsulated into a DFM which users can install on-demand.

Defect Scenarios: Figure 1 shows three steps that lead to installation of the *Encryption* DFM. First, the user clicks on an e-mail's menu—denoted by three dots—to access more options. Clicking on the "Encrypt" option initiates the installation of the *Encryption* DFM. If the module installation is successful, the user will see a dialog to enter a password and confirms the encryption. Other-

```
1  #click on the "Security Alert" email's menu icon
2  driver.find_element_by_xpath(path_to_icon).click()
3  #click on the "Encrypt" option
4  driver.find_element_by_xpath(path_to_encrypt).click()
5  #enter encryption password in the password field
6  driver.find_element_by_id(password_field_ID).
7       send_keys(password)
8  #repeat password
9  driver.find_element_by_id(password_confirm_field_ID).send_keys(password)
10 #click on "OK" button
11 driver.find_element_by_id(OK_button_ID).click()
```

Fig. 2. An Appium test, written in Python, to test the installation success scenario of Figure 1.

wise, if the installation request fails due to a contextual reason (Section 3), K9Mail goes to the initial screen without notifying the users about the failure or instructing them to take proper actions—potentially giving users the false impression that the e-mail is encrypted. Without a proper test to reveal the failure scenario, a developer may not detect this defect. We describe two possible dynamic delivery failures for this scenario, i.e., network disconnection [6] and insufficient storage [17]. We further discuss properties of the required tests to identify such failures.

Tests: Figure 2 shows an Appium [19] system test to validate the behavior of K9Mail during installation of the *Encryption* module. While this test can validate the proper implementation of K9Mail when the installation request is successful, it is unable to capture dynamic delivery failures occurring due to network disconnection or insufficient storage. Consequently, we need two additional tests to validate the proper handling of unsuccessful installation of the *Encryption* module. These new tests, shown in Figures 3 and 4, include adding system events to the test in Figure 2. The test in Figure 3 disconnects the network connection before clicking on the "Encrypt" option (line 4), making the installation request fail, and then re-connects the network connection (line 8) after clicking on the "Encrypt" option. The test then checks whether the desired alert dialog, notifying the user about the network connection failure, is displayed. On the other hand, the test in Figure 4 fills the storage of the device before clicking on the "Encrypt" option, making the installation request fail due to insufficient storage (lines 4–7), and then reverts it after clicking on the "Encrypt" option (lines 11 and 12). The test then checks if the desired dialog, asking the user to remove unnecessary files, is displayed. There are two main challenges to designing such tests:

(1) Installation request failures depend on the contextual settings in which a test is executed and cannot be achieved simply through GUI actions. To that end, developers need to inject additional system or GUI events into the test event sequences to induce a failure. In the test case of Figure 3, line 4 contains the system event that disables the network, making the installation request for the *Encryption* DFM fail due to a network error (Section 3, Equation (1)). To ensure that the change in the state of the network only impacts the DFM installation and not the rest of the test, the network connection is restored afterward.
(2) The position of the injected events in the test event sequence is critical. For example, if the developer disables the network connection at the very beginning (line 2 in Figure 2), the test cannot load the list of e-mails, cannot find the menu icon for the "Security Alert" e-mail, and thus fails without being able to execute the module installation, resulting in an invalid test. Similarly, if the network connection is disabled after initiating the installation request (line 4 in Figure 2), the test cannot observe the behavior of K9Mail when the installation request fails; therefore, it would not be a useful test for validating the *Encryption* DFM's behavior.

3 DEFECT MODEL

The previous section's defect scenario describes an installation request failure. Developers should be aware of all the different situations causing the installation failure, two of which are mentioned

```
#click on the "Security Alert" email's menu icon
driver.find_element_by_xpath(path_to_icon).click()
#disconnect network connection
driver.set_network_connection(0)
#click on the "Encrypt" option
driver.find_element_by_xpath(path_to_encrypt).click()
#restore network connection
driver.set_network_connection(1)
# check if network connection alert dialog is displayed
Assert.assertFalse(driver.find_element_by_id(Network_Alert_Dialog_ID).isEmpty())
```

Fig. 3. An Appium test, written in Python, to test the installation failure scenario of Figure 1 due to no network connection.

```
#click on the "Security Alert" email's menu icon
driver.find_element_by_xpath(path_to_icon).click()
#Filling the storage of the device with a huge file
args = {"command": "df","args": ["/sdcard", "|", "grep", "\'data\'", "|", "awk", "-F", "\' \'", "\'{
    print $4}\'"]}
bytesAvailable = driver.execute_script("mobile: shell", args)
args = {"command": "dd","args": ["if=/dev/zero", "of=/sdcard/deleteme", "bs=1B", "count=" + str(int(
    bytesAvailable))]}
result = driver.execute_script("mobile: shell", args)
#click on the "Encrypt" option
driver.find_element_by_xpath(path_to_encrypt).click()
#Deleting the huge file to free up the storage
args = {"command": "rm","args": ["/sdcard/deleteme"]}
driver.execute_script("mobile: shell", args)
# check if insufficient storage alert dialog is displayed
Assert.assertFalse(driver.find_element_by_id(Insufficient_Storage_Alert_Dialog_ID).isEmpty())
```

Fig. 4. An Appium test, written in Python, to test the installation failure scenario of Figure 1 due to insufficient storage.

in Section 2, and take appropriate actions to handle them. To identify the situations leading to an installation failure, we first need to describe the process through which a DFM is installed.

Once an Android app attempts to install a DFM, it creates an installation request that can go through multiple states, as demonstrated in Figure 5. First, at the *Requesting* state, the app store API—the app's runtime interface with the app store services—sends the installation request to the app store (e.g., Google Play [23]). In the *Pending* state, the request has been accepted by the app store operator, and the download should start momentarily. Through Android dynamic delivery, DFMs are downloaded as separate APK files, called *split APKs*. During download time, the request is in the *Downloading* state. Once the device completes the download, in case the app has SplitCompat[2] installed, it can immediately access the DFM to install it, leading the request to the *Installing* state. Otherwise, the request will go to the *Downloaded* state, where the DFM split APK is downloaded but cannot yet be installed. As soon as SplitCompat is installed in the app, the DFM can be installed in the background by the app store operator. Finally, if the installation finishes successfully, the request is in the *Installed* state. During the *Pending* or *Downloading* states, the installation request can be canceled, leading to the *Canceling* and *Canceled* states. After downloading, the Android device automatically installs the DFM, which can be uninstalled by the user later. For modules that are larger than 10 MB, the installation requires user confirmation (the *User Confirmation* state). If the user confirms the installation, the request follows the normal lifecycle. Otherwise, it goes to the *Canceling* and *Canceled* states [24].

An installation request may also terminate the normal lifecycle and go to any of the *Failed* states, which are denoted by dashed lines in Figure 5. Based on the root cause of the failure, developers need to handle it accordingly. For instance, if the request fails due to a *network error*, developers

[2]SplitCompat is an Android library class that enables immediate access to code and resources of the DFM split APKs.

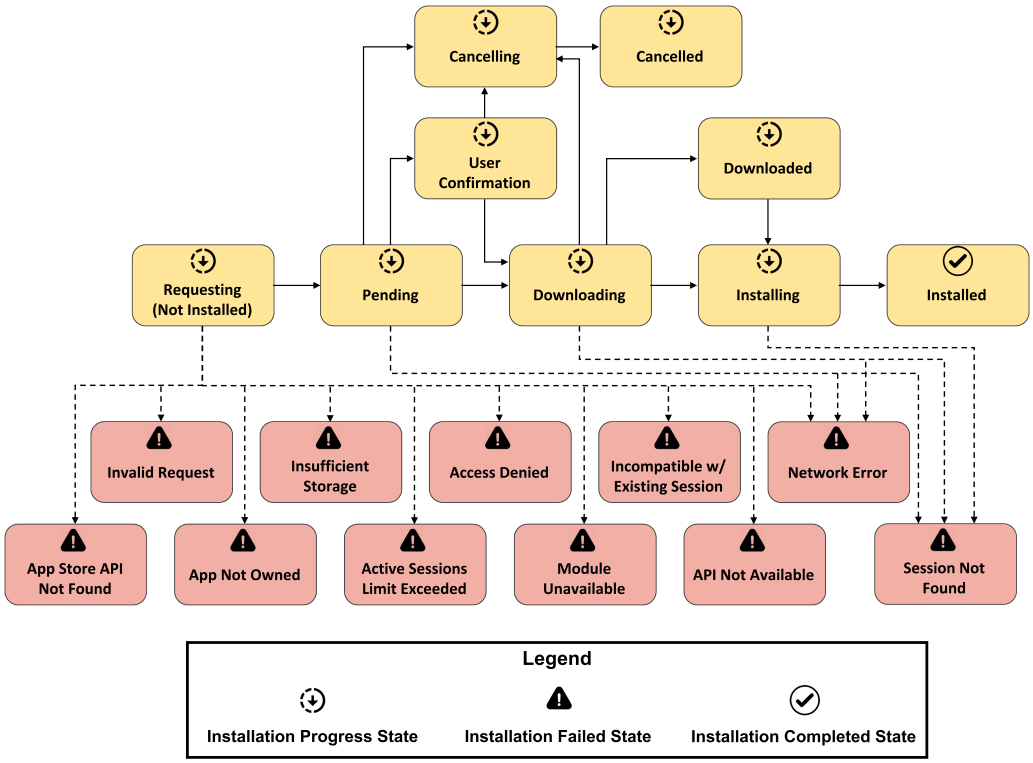

Fig. 5. The lifecycle of a DFM installation request.

should check the network connection and only resend the request if the network is stable. As another example, if the request fails due to *insufficient storage*, which prevents the module from being installed on the device, the developer should ask the user to free up some space, install the module, and monitor for different root causes of the module installation failure.

While Android documentation briefly explains the *Failed* states and provides recommendations for proper actions to handle the failure of an installation request [24], it does not sufficiently describe the context required to reach the *Failed* states for the purpose of testing. In fact, developers might have a hard time understanding the root cause of each *Failed* state [2–5, 7–9, 13–16]. For instance, [5] states, "My application uses app bundle, but dynamic feature performs a very poor rate of success when downloading, with a lot of error codes, and I cannot do anything for it. I search the docs but no help!". To overcome this challenge, we constructed a defect model for Android dynamic delivery failures. This defect model formally defines a *Failed State Context* as the context in which its associated *Failed* state will occur. The formulation will be used by DeltaDroid to augment existing tests to induce *Failed* states (Section 4).

To construct a comprehensive defect model, we explored multiple resources to understand the root causes of dynamic delivery failures and to identify contextual factors that trigger them. Specifically, we started from Android documentation and classified failures into 11 classes, based on the different error messages Android framework generates upon the occurrence of each failure [25]. We then explored issue repositories on GitHub [21] and discussion forums (e.g., StackOverFlow [26]) for keywords related to installation errors and DFM install failures, such as

dynamic feature module installation failure, *SplitInstallErrorCode*, names of each specific error code, and their messages included in the Android documentation [25]. Additionally, we investigated many open-source Android apps' implementations for instances of `SplitInstallErrorCode` and the way developers handle them using `onFailureListeners` upon the occurrence of different failures. Investigating these resources, we identified the contextual factors contributing to the manifestation of dynamic delivery. For a few cases where the mentioned resources were inconclusive, we exhaustively examined all the relevant contextual factors that could produce the desired *Failed* state.

3.1 Failed State Context Definition

The manifestation of a *Failed* state depends on the properties of an app bundle, DFM, installation request, and device configurations.

Definition 1. The configuration of a device in which a DFM will be installed can be formally specified as a tuple $Dev \equiv \langle Android_Version, App_Store, Account, Network, Storage \rangle$, where

- $Android_Version$ is the device's Android version,
- App_Store is the device's app store app (e.g., Play Store app),
- $Account$ is the user's app store account registered on the device,
- $Network \in \{true, false\}$ is the network connection status of the device (*true* if connected and *false* otherwise), and
- $Storage$ is the amount of storage available on the device.

Definition 2. An app bundle can be represented as a tuple $AB \equiv \langle DFMs, Permissions, DownloadSource \rangle$ where

- $DFMs \equiv \{DFM_i \mid i \in \{1, \ldots, n\}\}$ is a set of all DFMs in the app bundle AB,
- $Permissions$ is the set of permissions required to install DFMs, and
- $DownloadSource$ is the source from where the app bundle has been downloaded, e.g., Google Play Store.

Definition 3. A DFM can be formally specified as a tuple $DFM \equiv \langle AB, Min_SDK_Version, Authorized_Users, Size \rangle$ where

- AB is the app bundle of DFM,
- $Min_SDK_Version$ indicates the minimum API version of the Android device that the DFM is compatible with, i.e., *minSdkVersion* in DFM build file,
- $Authorized_Users$ are a set of users' app store accounts that have access to the DFM and can install it, and
- $Size$ is the necessary storage for installation of DFM.

Definition 4. An installation request, i.e., a request from an app to install a DFM is defined as a tuple $Req \equiv \langle SessionID, DFM, State \rangle$ where

- $SessionID$ is a unique identifier for a request assigned by the app store API to track the status of the request,
- DFM is the dynamic feature module that will be installed through this request, and
- $State \in \{R, P, UC, D_{ing}, D_{ed}, I_{ing}, I_{ed}, C_{ing}, C_{ed}, F\}$ is the current state of the installation request. As shown in Figure 5, the possible values are R (Requesting), P (Pending), UC (User Confirmation), D_{ing} (Downloading), D_{ed} (Downloaded), I_{ing} (Installing), I_{ed} (Installed), C_{ing} (Canceling), C_{ed} (Canceled), and F (Failed).

With these definitions, we can represent a Failed State Context as a tuple $FSC = \langle Dev, AB, Req \rangle$ and identify the contextual settings in which a distinct *Failed* state happens.

3.2 Failed States

As shown in Figure 5, an installation request may terminate in ten different *Failed* states: *Network Error, Insufficient Storage, Access Denied, Incompatible With Existing Session, Active Sessions Limit Exceeded, Module Unavailable, API Not Available, Session Not Found, App Not Owned*, and *Invalid Request*.

(1) Network Error. An installation request goes to the Network Error state if a network exception [6] occurs when the request is in any of the *Requesting*, *Pending*, or *Downloading* states. For a $dev : Dev$ and $req : Req$, the contextual parameters that induce such failure are as follows:

$$FSC_{Network_Error} \equiv req.State \in \{R, P, D_{ing}\} \ \wedge \ dev.Network_Connection = false. \quad (1)$$

To ensure developers have considered a proper action when such failure happens, there should be a test to disconnect network connection during the lifecycle of the request.

(2) Insufficient Storage. The Insufficient Storage failure happens when an app requests to install a DFM, whose size is larger than available storage on the phone. More specifically, once an installation request is initiated, the Android device checks its remaining storage before starting the installation request, and in case of insufficient available storage, it fails at the *Requesting* state. It will only start downloading and installing in case the device has enough storage. For a $dev : Dev$ and $req : Req$, the contextual parameters that induce such failure are as follows:

$$FSC_{Insufficient_Storage} \equiv req.State \in \{R\} \ \wedge \ (req.DFM.Size > dev.Storage). \quad (2)$$

Developers should handle this situation, e.g., show a proper notification informing the user about the space availability issue on the phone. To validate this behavior, a test needs to mock the storage of the device as being full.

(3) Access Denied. Without proper permissions to download a DFM, i.e., `REQUEST_INSTALL_PACKAGES` permission, an installation request from the app store operator is denied in its inception, i.e., when in the *Requesting* state. For an $ab : AB$ and $req : Req$, the contextual parameters that induce such failure are as follows:

$$FSC_{Access_Denied} \equiv req.State \in \{R\} \ \wedge \ Request_Install_Packages \notin ab.Permissions. \quad (3)$$

To ensure developers implement proper actions in response to this failure, test suites should have either a test that permanently revokes the app's `REQUEST_INSTALL_PACKAGES` permission or temporarily revokes it by taking the app to the background.[3]

(4) Incompatible With Existing Session. Installation of a DFM might be accessible from different paths in the program. Thus, more than one installation request can be sent to the app store operator for the same DFM. Specifically, consider the case where an installation request of a DFM is in progress and the DFM is not installed yet, and meanwhile, another installation request attempts to install the same DFM. In this case, the second request will terminate and reach the *Incompatible With Existing Session* state. This failed state happens while the first request is in any of the *Requesting*, *Pending*, *Downloading*, or *Installing* states, and the second request is in the *Requesting* state. For a $dev : Dev$, $req1 : Req$, and $req2 : Req$, the contextual parameters that induce such failure are

[3]From Android 11, an app's permission can be temporarily revoked while it is not in the foreground.

as follows:

$$\begin{aligned} &FSC_{Incompatible_With_Existing_Session} \\ &\equiv req_1.DFM = req_2.DFM \\ &\wedge\ req_1.SessionID \neq req_2.SessionID \\ &\wedge\ req_1.State \in \{R, P, D_{ing}, I_{ing}\} \ \wedge\ req_2.State \in \{R\} \end{aligned} \quad . \tag{4}$$

To ensure the app implements the proper means of accounting for this failure, a test should simultaneously initialize at least two installation requests for the same DFM.

(5) Active Sessions Limit Exceeded. While the previous situation includes multiple requests for the same DFM, an installation request for a DFM is also rejected if there is already an active installation request for another DFM. This failed state occurs while the first request is in any of the *Pending*, *Downloading*, or *Installing* states, and the second request is in the *Requesting* state. For a $dev : Dev$, $req1 : Req$, and $req2 : Req$ the contextual parameters that induce such failure are as follows:

$$\begin{aligned} &FSC_{Active_Sessions_Limit_Exceeded} \\ &\equiv req_1.DFM \neq req_2.DFM \\ &\wedge req_1.State \in \{P, D_{ing}, I_{ing}\} \ \wedge\ req_2.State \in \{R\} \end{aligned} \quad . \tag{5}$$

If the app contains multiple DFMs in the bundle, installing any of them may result in *Active Session Limit Exceeded* failed state. Since the implementation of these DFMs might be distributed in the code, different test cases executing different parts of the code are required to induce this failed state.

It is worth to note that *Active Session Limit Exceeded* state does not subsume *Incompatible with Existing Session*, as the *SessionID* of req_1 and req_2 might be the same in the former. To ensure developers implement proper actions to account for this failure, a test should simultaneously initialize installation requests for different DFMs.

(6) Module Unavailable. This failed state can occur due to two possible causes: (1) For each DFM installation request, the app store API checks whether the `minSdkVersion` of the DFM matches that of the Android system running on the device. *Module Unavailable* failed state occurs if the Android version running on the device is less than the `minSdkVersion` in the DFM's build file, i.e., mismatches the required API Version of the DFM. (2) In some cases, the developer might not make the DFM available to all users. If a DFM is unavailable for the user's app store account on the device, the request reaches the *Module Unavailable* state. This failed state happens while the request is in the *Requesting* state. For a $dev : Dev$ and $req : Req$, the contextual parameters that induce such a failure are as follows:

$$\begin{aligned} &FSC_{Module_Unavailable} \equiv req.State \in \{R\} \\ &\wedge (dev.Android_Version < req.DFM.Min_SDK_Version. \\ &\vee\ dev.Account \notin req.DFM.Authorized_Users) \end{aligned} \tag{6}$$

To validate the app implements proper means of handling this failure, a test suite should include a test that either changes the Android version of the device or the app store account on the device.

(7) API Not Available. App bundles and thus on-demand DFMs are available on Android version 5 (API level 21) or higher. For older versions, developers should change the configuration of DFMs, such that they can be downloaded with the base APK. This allows the app to support downloading all modules of an APK, similar to a monolithic APK from Android versions that do not support DFMs. Otherwise, users of devices with older versions can only download the base APK. Attempts of an app to violate this by trying to download a DFM results in the installation request terminating with the *API Not Available* error. This failed state occurs while the request is in the

Requesting state. For a $dev : Dev$ and $req : Req$, the contextual parameters that induce such failure are as follows:

$$FSC_{API_Not_Available} \equiv \ req.State \in \{R\} \wedge dev.Android_Version < Android_5. \quad (7)$$

A test to validate the proper action to account for this failure should mock the Android version running on the device.

(8) App Store API Not Found. This *Failed* state happens when the app store (e.g., Play Store app) is either not installed or not the official version. This failed state occurs while the request is in the *Requesting* state. For a $dev : Dev$ and $req : Req$, the contextual parameters that induce such failure are as follows:

$$FSC_{App_Store_API_Not_Found} \equiv req.State \in \{R\} \wedge dev.App_Store \neq Official_App_Store_app. \quad (8)$$

A test to validate an app's behavior in response to this failure should mock a device that does not have the official app store installed.

(9) Session Not Found. This *Failed* state happens when the session ID generated by the app store operator for an installation request is not valid, i.e., does not correspond to an active session in the app store services. This failed state occurs while the request is in any of the *Pending*, *Downloading*, or *Installing* states. For a $req : Req$, the contextual parameters that induce such failure are as follows:

$$FSC_{Session_Not_Found} \equiv req.State \in \{P, D_{ing}, I_{ing}\} \wedge req.SessionID = invalid. \quad (9)$$

(10) App Not Owned. This *Failed* state happens when the app has not been installed by the designated app store and the feature cannot be downloaded. This failed state occurs while the request is in the *Requesting* state. For a $req : Req$, the contextual parameters that induce such failure are as follows:

$$FSC_{App_Not_Owned} \equiv req.State \in \{R\} \wedge ab.DownloadSource \neq Official_App_Store_app. \quad (10)$$

(11) Invalid Request. This *Failed* state happens when the app store received the installation request, but the request is not valid., i.e., the information included in the request is not complete or accurate. This failed state occurs while the request is in the *Requesting* state. For an $ab : AB$ and a $req : Req$, the contextual parameters that induce such a failure are as follows:

$$FSC_{Invalid_Request} \equiv req.State \in \{R\} \ \wedge \ (req.SessionID = invalid \ \vee req.DFM = invalid)\,. \quad (11)$$

A test to validate an app's behavior in response to the last three failures should mock the app store operator API to generate an invalid session ID for an installation request, e.g., $SessionID = null$, or an invalid source of download for an app bundle.

4 DELTADROID

Figure 6 depicts a high-level overview of DeltaDroid, consisting of two major components: *Test-Suite Analysis* and *Test-Suite Augmentation*.

Test-Suite Analysis takes an Android app bundle and the corresponding initial test suite as inputs and produces the *Baseline Test Cases*—representing those tests that initiate a DFM installation—and the associated *Metadata*—data about the specific test step in which a DFM is installed. The *Baseline Test Cases* are subsequently reused in the creation of new tests.

Test-Suite Augmentation takes the *Baseline Test Cases* and associated Metadata together with the *Defect Model*, which formally defines the contexts in which a DFM installation request can fail, as inputs and generates additional tests that are effective for dynamic delivery validation. Specifically, this component creates new tests by first replicating a baseline test and then modifying it with a combination of system and GUI events to induce the context that may manifest installation request

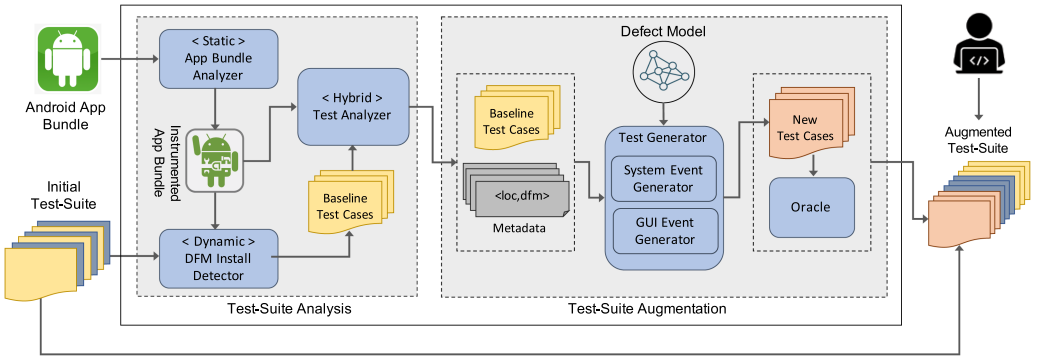

Fig. 6. A high-level overview of DeltaDroid.

```
1 //create an instance of SplitInstallManager.
2 SplitInstallManager splitInstallManager = SplitInstallManagerFactory.create(context);
3 //create a request to install a module.
4 SplitInstallRequest request = SplitInstallRequest.newBuilder().addModule("Encryption").
      build();
5 //submit the request to install the module
6 splitInstallManager.startInstall(request);
```

Fig. 7. Code snippet from K9Mail app that requests the installation of *Encryption* DFM.

failures. The newly generated tests create all the failure scenarios that an app should handle. In the remainder of this section, we provide a more detailed explanation of the components comprising DeltaDroid.

4.1 Test-suite Analysis

The *Test-Suite Analysis* component identifies tests that are suitable for validating the behavior of dynamic delivery in the app. To that end, *App Bundle Analyzer* pinpoints the code responsible for installing DFMs in the app and instruments it. Subsequently, *DFM Install Detector* executes the initial test suite on the instrumented app to determine the *Baseline Test Cases*, i.e., the tests in the initial test suite that execute the installation request. Finally, *Test Analyzer* identifies the locations in baseline tests where additional GUI or system events should be injected to create effective dynamic delivery tests.

4.1.1 App Bundle Analyzer. To aid in understanding how *App Bundle Analyzer* works, Figure 7 shows the code snippet corresponding to the installation request of the `Encryption` DFM in k9Mail. The code initiates the request by creating instances of `SplitInstallManager` (line 2) and `SplitInstallRequest` (line 4). It then sends the installation request of the `Encryption` DFM to the app store API by invoking the `startInstall` method (line 6).

App Bundle Analyzer performs a flow-sensitive analysis to find invocations of `startInstall` method and injects a logging statement immediately before it in the control-flow graph. Additionally, *App Bundle Analyzer* registers a `SplitInstallStateUpdatedListener` to monitor and log the state of the installation request at run-time.

4.1.2 DFM Install Detector. *DFM Install Detector* takes the instrumented app bundle as input, runs it against the initial test suite, and identifies the tests that initiate DFM installation requests as *Baseline Test Cases*, i.e., tests that execute the instrumented log statements.

ALGORITHM 1: *Test Analyzer*

```
Input: app // Instrumented app bundle
Input: T // Baseline tests
Output: MD: {md | md = ⟨location, DFM⟩} // Baseline tests' metadata
1  MD ← ∅
2  foreach test ∈ T do
3      loggedEntries ← RunTestCase(test, app)
4      sortByTimeStamp(loggedEntries)
5      foreach entry ∈ loggedEntries do
6          if entry.type is TestEvent then
7              location ← getEventID(entry)
8          else
9              DFM ← getDFMIDentifier(entry)
10             MD ← MD ∪ ⟨location, DFM⟩
11             break
12         end
13     end
14 end
```

4.1.3 Test Analyzer. After identifying *Baseline Test Cases*, the next step is to find the proper location of modification, i.e., the index in the test event sequence where new test events should be injected, which we refer to as *target location*. To that end, *Test Analyzer* first instruments the tests to record all the test events, their types, and orders. Events could be either GUI events (e.g., button clicks) or system events (e.g., changes in network connection). It then uses Algorithm 1 to locate the last event in each test after which an installation request is initiated, i.e., the *target location*.

Algorithm 1 shows how *Test Analyzer* identifies target locations. *Test Analyzer* takes the instrumented *app* and a set of baseline tests, T, as input, and provides a set of tuples $\langle$ *location*, *DFM*$\rangle$ as output. The first element in the output tuple, *location*, indicates the index at which a $test \in T$ should be modified, and the second element indicates the *DFM* that is installed by *test* at that *location*.

Algorithm 1 starts by initializing the metadata, *MD*, to an empty set. For each instrumented test $test \in T$, Algorithm 1 executes the test on the instrumented app bundle, *app*, and collects all the entries recorded in the log file during test execution in *loggedEntries* (line 3). These entries are either test events that are recorded by the instrumented test, *test*, or log messages that show *app* has initiated an installation request. At the next step, the Algorithm sorts the logged entries based on their corresponding time stamp, i.e., the actual time the event happens, to account for potential race conditions (line 4). Subsequently, Algorithm 1 iterates over all the entries in *loggedEntries* to identify the target location, i.e., the location of the last event in each baseline test after which an installation request is initiated (lines 5–13). Specifically, it starts from the first entry in *loggedEntries* and keeps track of the order of test events (lines 6 and 7). Once Algorithm 1 reaches an entry that is made by *app* that installs a DFM (line 8), it extracts the DFM identifier from the entry (line 9) along with the location of the last test event, *location*, saves it as a tuple in *MD* (line 10) for the corresponding test.

The algorithm terminates once the target locations corresponding to all tests are identified. As a result, it provides a set of tuples containing the information required by the next component in DeltaDroid's workflow, i.e., *Test-Suite Augmentation*.

ALGORITHM 2: *Test-Suite Augmentation*

Input: FSCs: $\{fsc \mid fsc = \langle Dev, AB, Req \rangle\}$ `// Failed State Context formula`
Input: T `// Instrumented baseline tests`
Input: MD: $\{md \mid md = \langle location, DFM \rangle\}$ `// Metadata corresponding to baseline tests`
Output: GenTCs

```
1 GenTCs ← ∅
2 foreach test ∈ T do
3     foreach fsc ∈ FSCs do
4         events_sys ← systemEventGenerator(fsc)
5         events_GUI ← GUIEventGenerator(fsc)
6         test′ ← test ∪ injectEvents(md, events_sys, events_GUI)
7         GenTCs ← GenTCs ∪ test′
8     end
9 end
```

4.2 Test-suite Augmentation

Test-Suite Augmentation takes our defect model (Section 3) as well as the output of *Test-Suite Analyzer* as inputs and generates effective dynamic delivery tests as output, i.e., 11 additional tests for a given baseline test. Each of these new tests is responsible for creating one of the *FSC*s formally defined in Section 3 to make the installation requests fail. *FSC*s can be created by injecting additional system and GUI events into the test event sequence.

Algorithm 2 shows how *Test-Suite Augmentation* generates effective dynamic delivery tests. There are three inputs to this algorithm. The first input is $FSCs = \{fsc \mid fsc = \langle Dev, AB, Req \rangle\}$, which is a set of *Failed State Contexts* we formally defined in Section 3.2. The second and third inputs are *T* and *MD*, respectively, the output of *Test-Suite Analysis*. *T* represents a set of candidate tests and *MD* is the corresponding metadata, i.e., information about the DFM each test installs and the location at which DFM installation occurs within each baseline test.

Algorithm 2 initializes *GenTCs*, a set of newly generated tests, as an empty set (line 1). It then iterates over baseline tests (lines 2–7) and for each DFM installation request by a baseline test, it generates 11 new tests, where each of them is responsible to create the context corresponding to one of the FSCs (lines 3–6). Specifically, Algorithm 2 generates a set of test steps for creating system events (line 4) and GUI events (line 5), and injects them in the baseline *test* to create a new test, *test′*, (line 6). After generation of each new test, Algorithm 2 updates *GenTCs* with the new test (line 7).

The `systemEventGenerator` method is responsible for the generation of proper system events. It takes the formulation of a *Failed State Context*, *fsc*, parses the formulation to obtain contextual properties that contribute to the corresponding *fsc*'s failure, and generates system events accordingly. For example, to generate the example test for k9Mail (Figure 3), `systemEventGenerator` parses $FSC_{Network_Error}$ and identifies the state of the request, $req.State$, and connectivity status of the device, $dev.Network_Connection$, as the related contextual parameters (Equation (1)) and generates corresponding system events. There are two categories of system events that `systemEventGenerator` injects into the baseline tests. The first category includes system events that directly change a context, e.g., disabling the network connection, slowing down the network speed, or creating a large file to take up device storage. The second category includes system events that indirectly change contexts by simulating the device's running Android version, user's app store account, or the *SessionID* assigned to an installation request.

For a subset of FSCs in which the failure originates in the app store API (Equations (6)–(11)), `systemEventGenerator` mocks the app store API to induce a specific behavior. For instance, in *Module Unavailable FSC*, the failed state will be induced if the version of the requested module is not compatible with the app bundle uploaded in the app store. To that end, `systemEventGenerator` replaces `SplitInstallManager` with a customized implementation reflecting the required context.

System events and mocking are not enough for creating a subset of FSCs. For instance, in the case of *Access Denied* (Formula 3), an effective test should revoke the `REQUEST_INSTALL_PERMISSION` of the app. This can be done either by permanently revoking the permission through Android settings or temporarily revoking the app's permission to download a DFM by moving the app to the background right before the DFM installation begins. Each of these decisions requires a series of consecutive GUI actions. For example, one series of GUI actions may involve clicking on the *Home* button to move the app to the background, relaunching the app in the same state by clicking the *App Switch* button, and choosing the app among the recent running apps.

After producing the proper events, Algorithm 2 injects them into a copy of the given baseline test and generates a new test (line 6). To that end, the `injectEvents` method takes $event_{sys}$ and $event_{GUI}$ sets and injects them into *test* at the index specified by *md.location*. Specifically, `injectEvents` first injects $event_{sys}$ and then $event_{GUI}$. This is mainly because some GUI actions are useless without proper system events before them. For example, in the case of *Active Session Limit Exceeded*, the installation request should be in the *Pending*, *Downloading*, or *Installing* states *before* requesting installation of another DFM to make the failure occur. To ensure this, we need to inject system events that slow down the speed of network connection, making the transition of installation requests between states slower and allowing GUI actions to be effective. In the rare cases where the baseline tests already contain proper events to simulate any or a subset of FSCs, the specific FSC(s) are simply repeated. This repetition does not break the augmented test.

Finally, once a new test is generated, Algorithm 2 adds it to the set of newly generated tests. Algorithm 2 terminates after generating all the tests that make each installation request enter all the 11 *Failed* states described in Section 3. That is, the size of *GenTCs* is 11 times the size of baseline tests, i.e., $|GenTCs| = 11 \times |T|$.

Once the new tests are generated, the *Oracle* component automatically determines if developers properly handle the failed states. We follow the specification of *unhandled* behavior from Android documentation [25], in which an app takes no action, visible to the user, to handle the failure. Such behavior in our illustrative example (Section 2) could be due to either missing an `OnFailureListener` in the code or any additional UI element on the app screen notifying the user about a failure. To identify *unhandled* behaviors, the oracle performs two actions. First, it investigates the invocation of `OnFailureListener` during the execution. If such a callback is not invoked after an installation failure, a clear indicator is that a DFM failure is not handled in the implementation. However, a developer may implement `OnFailureListener` in the code without actually handling the failure in a meaningful manner. To account for such cases, our proposed oracle investigates changes in the visible UI elements *before* and *after* the DFM installation failure. If there is an addition of UI elements to the screen, e.g., a UI widget is added to show an error message, developers have satisfied the minimum requirement to handle the DFM failure, i.e., informing users about the failure [25]. Otherwise, the oracle can confidently judge that the developer has not properly handled the DFM installation failure.

We do not generate assertions here for two reasons: First, the purpose of DFM testing is to help developers with the steps that lead to dynamic delivery failures and determine whether a particular failed state was handled properly. Assertions for dynamic delivery testing are highly app-specific. For example, depending on the developer's design decision, they can either show a message, which notifies the user about the failure and its reason, or stops the installation progress

bar. In the former case, the assertion checks for the visibility of a dialog message, e.g., line 10 in Figure 3. In the latter, the assertion could check for a specific amount of progress (e.g., 0%) on the progress bar. As another example, the developer of k9Mail, in our illustrative example (Section 2), may decide that the proper action to handle *Network Failed* state is to notify the users and ask them to retry once the network is connected. Hence, the appropriate assertion should check the existence of the notification element. Alternatively, the developer may decide to flag the encrypted e-mails with a specific icon so that the user knows if an e-mail is encrypted or not. In this design decision, the appropriate assertion should check the existence of the specific icons. While these assertions are simple, they require the developer's judgment and understanding of the app's behavior. That said, DeltaDroid can suggest the locations where developers could add assertions.

5 EVALUATION

To evaluate DeltaDroid's effectiveness in validating Android apps' dynamic delivery, we investigate the following research questions:

- **RQ1:** *Effectiveness of Inducing Failed States.* How effective is DeltaDroid in augmenting test suites to generate test cases that induce different *Failed* states?
- **RQ2:** *Effectiveness of Revealing Installation Defects.* To what extent does DeltaDroid reveal installation defects in real-world Android apps' behavior?
- **RQ3:** *Performance.* What is DeltaDroid's runtime efficiency in terms of execution time?

5.1 Experimental Setup

To answer these research questions, we selected a set of open-source Android apps from GitHub [21] which are also uploaded to Google Play. Our inclusion criteria select apps that (1) are open-source, (2) implement Android dynamic delivery, (3) have at least two DFMs, and (4) have an initial test suite. Note that dynamic delivery features need the developer to sign an app when deployed onto the Google Play platform which, in turn, requires our subject apps to be open source. To select subject apps that implement dynamic delivery, we searched for "`apply plugin: com.android.dynamic-feature`" statement in the Android projects' build files, i.e., Gradle [22]. Among the search results, we excluded toy or demo apps, which are minimal Android apps with no specific functionalities, mainly developed to learn or examine Android dynamic delivery. In the next step, we excluded apps with only one DFM, as one of the *Failed* states, namely *Active Session Limit Exceeded*, requires subjects to have at least two DFMs.

Our final dataset includes 25 apps with a total of 51 DFMs, shown in Table 1. These apps come from various categories, reducing any bias towards a certain type of app. Among these 25 apps, the initial test suite of 15 apps has tests to cover all of the DFM installation requests in the code. Since DeltaDroid requires baseline test cases that attempt to install DFMs, we manually extended the test suites of the remaining 10 apps with a total number of 20 test cases that initiate the installation of DFMs. The additional test cases are small, i.e., only 10–22 lines of code per test case.

For a thorough evaluation of DeltaDroid, we first compare it against a baseline static analysis approach, which can detect failures corresponding to the missing implementation of `OnFailureListener` callbacks using Soot [52]. This approach performs a flow-sensitive analysis to find locations in code where an installation request is initiated. It then checks whether the `OnFailureListener` is implemented. In case the `OnFailureListener` callback is not implemented, the static analysis will report a dynamic delivery failure. The rationale here is that without an `OnFailureListener` callback implementation, the app will fail in any of the *Failed State Contexts*. The issue with the static analysis approach is that it does not execute the `OnFailureListener`

Table 1. Subject Apps

No.	Application Name	Application Category	# DFMs	Size (LoC)	Test-Suite Size Initial	Aug.
1	Alert	Productivity	2	914	12	34
2	Amaze File Manager	Utility	3	107,141	68	101
3	AntennaPod	Entertainment	2	192,094	51	73
4	K9Mail	Utility	2	218,242	95	117
5	LeafPic	Utility	2	144,332	16	38
6	AnkiDroid	Educational	2	113,547	32	54
7	Authorizer	Utility	2	92,990	22	44
8	Open Camera	Utility	2	73,128	11	33
9	TimeTable	Educational	2	39,371	17	39
10	Ping	Utility	2	1,633	13	35
11	Bills	Utility	2	55,304	12	34
12	Travel Destinations	Lifestyle	2	762	12	34
13	Alkaa To-Do	Productivity	2	46,064	37	59
14	Kredit	Utility	2	414	12	34
15	Calculator	Productivity	2	36,884	12	34
16	Movie Stats	Entertainment	2	61,646	10	32
17	Mediation Ads	Lifestyle	2	29,482	4	26
18	MyNews	Educational	2	41,764	4	26
19	Dictionary	Educational	2	32,841	16	38
20	JackOfAll	Productivity	2	37,958	6	28
21	Income Tracker	Productivity	2	31,025	19	41
22	Translator	Utility	2	131,401	12	34
23	BigFiles	Utility	2	213,679	8	30
24	SuperHero	Lifestyle	2	1,784	10	32
25	TOKO Game	Game	2	48,491	10	32

callback, and, as a result, it does not provide the developers with accurate information regarding their app's behavior under different *Failed State Contexts*.

We also compared DeltaDroid against three alternative testing approaches. The first one is Monkey [18], one of the most widely used automated testing tools for Android in practice, which is shown to outperform many other automated testing tools [36]. Second, we compared DeltaDroid against APE [44], the state-of-the-art automated model-based approach for testing Android apps, which dynamically optimizes the model based on the run-time information. APE is shown to outperform similar state-of-the-art Android GUI testing tools (*Sapienz* [56] and *Stoat* [66]) in terms of both testing coverage and the number of detected unique crashes. Third, we compared DeltaDroid against App Crawler [28], which tests an Android app by running alongside the app and automatically issuing actions, e.g., tap, swipe, and so on, to explore the state-space of the target app. It automatically terminates when there are no more unique actions to perform, the app crashes, or after a designated timeout.

Test augmentation techniques target a specific goal in testing, e.g., achieving higher coverage of the code or reproducing the crashes. Unlike those techniques, DeltaDroid uses test augmentation to cover all the failed states in the lifecycle of a DFM installation request by relying on a novel defect model. None of the other test augmentation approaches target the same goal. Hence, they

are not suitable candidates for comparison against DeltaDroid. That said, we tried to compare DeltaDroid with Thor [31], a test suite augmentation technique in the domain of Android, since it creates events that can induce the *Network Error* and *Access Denied* failed states (but not other failed states). However, the current version of Thor is outdated, as also mentioned in their issue repository [29]. We have also contacted the authors who confirmed that Thor is no longer actively maintained and, since it is built on Android 4, it will not support Android apps with dynamic delivery.

In addition, we were unable to compare our results against *TimeMachine* [41], since the current version of it does not support Google Play Services, as reported on its issue repository [30], preventing us from running it on app bundles that use dynamic delivery.

Note that no other testing tool is quite comparable to DeltaDroid, as none of those tools generate the necessary system events for testing Android dynamic delivery. Explicitly handling dynamic delivery of Android apps (e.g., DFMs), and representing and accounting for the corresponding defect model and *Failed State Contexts*, are key novel elements of DeltaDroid that contribute to its testing effectiveness.

For the purpose of comparison, we gave Monkey and APE an hour to run, similar to prior studies [36, 44]. We also ran App Crawler without any designated timeout to be able to explore all states of the application. To further avoid any bias in favor of DeltaDroid, we re-installed the apps on the device once any test by Monkey, APE, and App Crawler completed a DFM installation. That is mainly due to the fact that once a DFM is installed, a *Failed* state cannot possibly be reached. Therefore, alternative approaches had many chances to cover *Failed* states during the hour-long period.

5.2 RQ1: Effectiveness of Inducing Failed States

To answer RQ1, we ran DeltaDroid over all the subject apps to augment their existing test suites. Table 1 shows the size of the initial and augmented test suites for each subject app under the *Test-Suite Size* column. These results confirm that DeltaDroid was able to generate a new test case for each *Failed* state for subject apps' DFMs, i.e., the size of the initial test suite is increased by $\#DFM \times 11$ tests.

We executed augmented test suites and ran Monkey, APE, App Crawler, and the baseline static analysis approach on the subject apps. Each app is instrumented (Section 4.1.1); we thus monitored the apps' log statements during test execution to identify whether the tests covered different *Failed* states. If the log record of a test indicates that the installation request entered any of the *Failed* states, we confirm that the test covers the state. We also ran the applications' initial test suite without manually extending them, and they could not test any of the defined failed states, as these states occur only under peculiar contextual settings.

Table 2 demonstrates the results of this study. It shows that DeltaDroid was able to reach almost all of the *Failed* states for the DFMs in a total of 25 subject apps. The only exception was the *Access Denied Failed* state, which was only reached by the tests in one app. The scenario in which an installation request can enter the *Access Denied Failed* state is quite a rare case that requires the app to initiate the installation from a concurrent thread that can be executed while the app is in the background (see Equation (3)). Only one of the apps among our subjects, namely *Travel Destinations*, was implemented in such a way, for which DeltaDroid was able to cover the *Access Denied Failed* state.

Monkey, APE, and App Crawler were all able to generate tests that installed all the DFMs in the subject apps, as reported in Table 2. However, they all performed quite poorly in terms of testing different *Failed* states. Specifically, APE was able to only cover the *Active Sessions Limit Exceeded Failed* state for 1 DFM. App Crawler performed better than APE and covered two *Failed* states,

Table 2. Testing Tools' Effectiveness of Covering *Failed* States

	Monkey [18]			APE [44]			App Crawler [28]			Baseline Static Analysis		DeltaDroid		
Failed State	# Apps	# DFMs	Avg. T.T.F.[1]	# Apps	# DFMs	Avg. T.T.F.	# Apps	# DFMs	Avg. T.T.F.	# Apps	# DFMs	# Apps	# DFMs	Avg. T.T.F.
N.E.	19	38	611	0	0	-	-	-	-	9	19	25	51	7
I.S.	0	0	-	0	0	-	-	-	-	9	19	25	51	9
A.D.	0	0	-	0	0	-	-	-	-	1	1	1	1	13
I.W.E.S.	0	0	-	0	0	-	1	1	3	9	19	25	51	5
A.S.L.E.	4	5	193	1	1	221	2	2	2	9	19	25	51	6
M.U.	0	0	-	0	0	-	-	-	-	9	19	25	51	4
A.N.A.	0	0	-	0	0	-	-	-	-	9	19	25	51	5
A.S.A.N.F.	0	0	-	0	0	-	-	-	-	9	19	25	51	5
S.N.F.	0	0	-	0	0	-	-	-	-	9	19	25	51	4
A.N.O.	0	0	-	0	0	-	-	-	-	9	19	25	51	4
I.R.	0	0	-	0	0	-	-	-	-	9	19	25	51	6
Avg. Total time per app	3,600 seconds			3,600 seconds			52 seconds			53 seconds		146 seconds		
% Installed DFMs	100%			100%			100%			N/A[2]		100%		

[1]T.T.F., short for Time To Failed states, indicates the testing time required to reach the failed state.
[2]The baseline static analysis approach does not initiate DFM installations, since it statically analyzes the apps.

i.e., *Incompatible With Existing Sessions* in 1 DFM of 1 app and *Active Sessions Limit Exceeded* in 2 DFMs of 2 different apps. While Monkey performed better than APE and App Crawler in covering *Failed* states, it was far behind DeltaDroid. That is, Monkey was able to reach the *Network Error Failed* state for 38 DFMs of 19 apps and the *Active Sessions Limit Exceeded Failed* state in 5 DFMs of 4 apps, making it cover 2 *Failed* states and 43 DFMs in total.

The baseline static analysis approach was able to inform us about missing the implementation of `OnFailureListener` callbacks in 19 DFMs of 9 apps, indicating the possibility of all types of Failed states for these apps. However, the static analysis approach results in a false negative in case the `OnFailureListener` is implemented, but not in a way to properly handle the failed states. Additionally, static analysis cannot generate tests to help developers induce the failed state contexts.

Overall, Table 2 demonstrates that DeltaDroid significantly outperforms alternative approaches in testing Android dynamic delivery. Additionally, these results confirm our choice to formulate dynamic delivery testing as a test-suite augmentation technique, since even a pure random testing technique, such as Monkey, can reach program points that install DFMs.

5.3 RQ2: Effectiveness of Revealing Installation Defects

To assess the DeltaDroid's ability to reveal defective behavior in Android apps, we investigated each app's behavior in our dataset once any of the *Failed* states occur. Proper behavior for an app entering a *Failed* state could notify the users of the situation, explain the reason for failure, give them an instruction, or retry the installation as described in Android documentation [27]. However, in our study, we conservatively consider a defective behavior only when an app crashes (*Crash*) or takes no action to handle the failure, i.e., no change in the behavior of the app (*Unhandled*), such as the illustrative example in Section 2. As a result of this study, we found 18 apps (i.e., 72%) with a total of 160 unhandled behaviors upon the occurrence of a *Failed* state. More importantly, we discovered 48 new instances of the app crashing in 7 apps (i.e., 28%), which were not detected by their initial test suites. Unlike a typical crash where simply restarting the app is sufficient to return the app to a stable state, in all of these 7 crashing apps, we had to manually uninstall and install the apps to resume their regular functionalities. We believe this is because DFM-induced crashes occur during the installation of new modules that, if not completed properly,

Table 3. Installation Defects Revealed by DeltaDroid

No.	Application Name	Failed States										
		N.E.	I.S.	A.D.	I.W.E.S.	A.S.L.E.	M.U.	A.N.A.	A.S.A.N.F.	S.N.F.	A.N.O.	I.R.
1	Alert	UH	UH	-	UH	CR	UH	UH	UH	UH	UH	UH
2	Amaze File Manager	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
3	AntennaPod	✓	✓	-	✓	✓	✓	✓	✓	✓	✓	✓
4	K9Mail	UH	CR	-	CR	CR	CR	CR	CR	CR	CR	CR
5	LeafPic	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
6	AnkiDroid	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
7	Authorizer	✓	✓	-	✓	✓	✓	✓	✓	✓	✓	✓
8	Open Camera	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
9	TimeTable	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
10	Ping	UH	UH	-	CR	CR	UH	✓	✓	✓	✓	✓
11	Bills[1]	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
12	Travel Destinations	✓	✓	✓	CR	✓	✓	CR	CR	CR	CR	CR
13	Alkaa To-Do	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
14	Kredit	✓	✓	-	UH	UH	✓	UH	UH	UH	UH	UH
15	Calculator	CR	CR	-	CR	CR	CR	CR	CR	CR	CR	CR
16	Movie Stats	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
17	Mediation Ads	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
18	MyNews	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
19	Dictionary	CR	CR	-	CR	CR	CR	CR	CR	CR	CR	CR
20	JackOfAll	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
21	Income Tracker	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
22	Translator	CR	CR	-	CR	CR	CR	CR	CR	CR	CR	CR
23	BigFiles	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
24	SuperHero	✓	✓	-	✓	✓	✓	✓	✓	✓	✓	✓
25	TOKO Game	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
% of apps not handling the failed state		64%	64%	0%	64%	60%	64%	64%	64%	64%	64%	64%
% of apps crashing in the failed state		16%	16%	0%	24%	24%	16%	20%	20%	20%	20%	20%

[1]Bills app freezes in every Failed state in which the user cannot have any further interaction unless the app is restarted or forced to quit.

corrupt the app bundle permanently. DeltaDroid's oracle was able to successfully determine the crashes, handled, and unhandled behaviors in 23 apps in our dataset (i.e., 92%). For the remaining 2 apps, the oracle resulted in false positives, i.e., incorrectly identified an unhandled behavior as handled since a UI element appeared on the screen after the failure, but not for the purpose of handling it.

Table 3 describes the apps' detailed behavior in our dataset under different *Failed* states. Notations "UH" (colored in yellow) and "CR" (colored in red), respectively, imply unhandled behaviors and crashes under the *Failed States* columns. The checkmark notation (colored in green) indicates that the app handles the corresponding *Failed* state by taking an extra action. All of the 6 apps in our dataset that handle at least one *Failed* state only display an error message without providing any instructions to the user. Excluding the rare scenario of *Access Denied Failed* state, our evaluation results indicate that for different *Failed* states, the unhandled behavior ranges from 60% to 64%, and crashes range from about 16% to 24% of the apps. These results emphasize the importance of testing Android dynamic delivery and the consequences of disregarding it.

One important outcome of these results is that a mere static analysis is not effective for validating dynamic delivery. Specifically, static analysis techniques, e.g., the baseline static analysis approach, at best, can only determine dynamic delivery failures corresponding to the missing implementation of "OnFailureListener" callback. Such failures constitute 70 out of 160 unhandled behaviors and 10 out of 48 crashes in our results. This is because dynamic delivery failures depend on contextual settings, e.g., regarding network connection, permissions, device storage, and so on that require app execution and dynamic analysis.

Finally, we reported all the discovered issues to the subject app developers. As of this submission date, 21 crashes and 54 unhandled behaviors are confirmed by the developers. Developers' responses also confirm the wide-reaching importance of the problem, the difficulty of testing Android dynamic delivery, and their interest in using our technique. For example, one developer indicated that they had not considered testing the installation failure scenarios before our report.

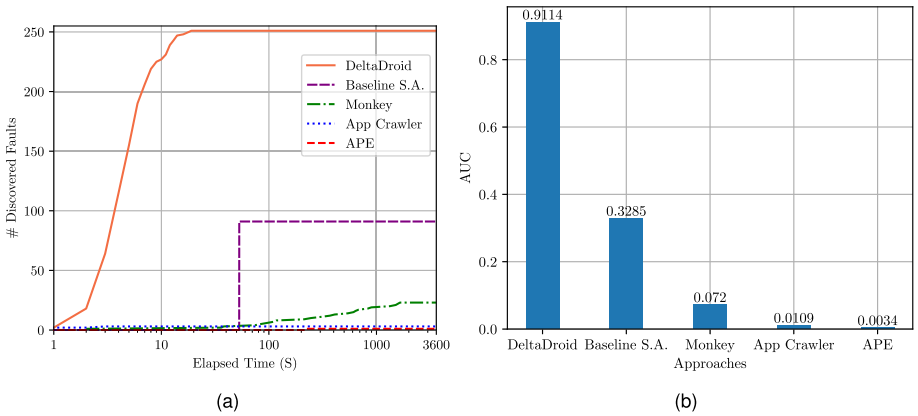

Fig. 8. The results of comparing alternative approaches in revealing dynamic delivery faults over time. (a) The number of discovered faults over the elapsed time for all approaches listed in Table 2. (b) The AUC of the plots in Figure 8(a).

The developer said, "As a developer, we try to cover all the failure scenarios, and testing them is not straightforward. Therefore, a testing tool that could detect all the loopholes for dynamic delivery implementation would be very helpful to us." Another developer noted that "The Android documentation is not very clear, and it is indeed difficult to test traditionally, or it requires a lot of code. I think a framework or utility to check the behavior of Android apps when module installation fails could be very useful".

5.4 RQ3: Performance

To investigate the performance characteristics of DeltaDroid, we first assess the performance of its two main components, i.e., *Test-Suite Analysis* and *Test-Suite Augmentation*. We next compare the efficiency of DeltaDroid to alternative approaches. We ran all of the tools on a Pixel 3a mobile device with Android 10 and an Android Emulator with Android 9 running on a MacBook Pro 2013 (2.3 GHz Intel Core i7, 16 GB, MacOS 10.14), depending on the Android versions of the apps.

To assess the performance characteristics of DeltaDroid, we measured the execution time corresponding to each of its components, as shown in Table 4. On average, it takes about 69 seconds for DeltaDroid to perform all of the required analysis on the initial test suite and 8 seconds to generate new test cases. The overall average execution time of 77 seconds confirms the scalability of DeltaDroid for validating Android apps' dynamic delivery.

To investigate the efficiency of DeltaDroid compared to alternative approaches, listed in Table 2, we measured the number of discovered dynamic delivery faults over a specific amount of time for each approach. To that end, we considered the maximum execution time of all approaches, i.e., an hour for Monkey and APE. Figure 8(a) shows the resulting plots with each approach in a different color. Then, to better compare the effectiveness of all candidate approaches, we measured the corresponding **area-under-the-curve** (**AUC**) for each plot. Figure 8(b) shows the amount of AUC of the mentioned plots for each approach. The results indicate that DeltaDroid's efficiency in revealing dynamic delivery faults is exceedingly superior to alternative approaches.

Furthermore, to compare DeltaDroid's efficiency with alternative approaches, we measured the average time to cover a *Failed* state for the first time in each testing approach, i.e., Monkey, APE, and App Crawler. This metric does not apply to the baseline static analysis approach since it statically analyzes the apps' source code, reports the final results, and terminates the execution.

Table 4. Execution Time of DeltaDroid

Phase	Avg. Execution Time (s)
Test-Suite Analysis	68.94
Test-Suite Augmentation	8.25
Total	77.19

Table 2 shows the results of this study (under columns *Avg. T.T.F.* which is short for Time to Failed state). DeltaDroid takes about 6 seconds on average to reach a *Failed* state, with a minimum of 4 and a maximum of 13 seconds. The corresponding numbers for Monkey, APE, and App Crawler are 402, 221, and 2.5 seconds on average, respectively. Although App Crawler reaches the failed states exceedingly fast compared to DeltaDroid, it performs quite poorly in terms of covering different failed states.

Furthermore, we calculated the average testing time of each technique and the total execution time for the baseline static analysis approach among all subjects. The results show that the test cases generated by DeltaDroid took a total time of 146 seconds, on average, to test each app. On the other hand, Monkey and APE were allowed to run for an hour on each app. App Crawler completed testing each app after 52 seconds, on average. The events DeltaDroid injects, e.g., connecting and disconnecting Wi-Fi for the Network Error failed state, or filling the device's remaining storage for the Insufficient Storage failed state, take more time compared to GUI events, e.g., clicking buttons. Therefore, DeltaDroid is slightly slower compared to App Crawler. The baseline static analysis approach also took a total time of 53 seconds, on average, to execute on each app. Although the App Crawler and the baseline static analysis approach execute more quickly than other techniques, these faster techniques are substantially inferior to DeltaDroid in terms of covering different failed states and revealing dynamic delivery defects.

6 THREATS TO VALIDITY

The main threat to internal validity is that DeltaDroid, indeed as a test augmentation technique, requires initial tests that reach program points that install DFMs. In case the test suite does not include test cases initiating the installation of DFMs, DeltaDroid reports no baseline tests to augment. Therefore, DeltaDroid would not be able to generate new test cases to test Android dynamic delivery failures. However, these test cases are easily obtained. According to our investigation, even a pure random testing technique such as Monkey [18] was able to create such tests for all apps, as reported in Table 2. For the evaluation of DeltaDroid, we have also manually extended the existing test suites of a subset of subject apps with test cases that attempt to install their DFMs, and reported that these test cases were small, i.e., only 10–22 lines of code per test case, and would be straightforward for a developer to write.

One of the main threats to external validity is the selection of subject Android apps in our evaluation. To mitigate this threat, we selected open-source Android apps that (1) implement Android dynamic delivery, (2) have at least two DFMs, and (3) have an initial test suite among the hundreds of applications on GitHub [21], one of the largest and most widely used open-source repositories online. Another threat to external validity is whether the types of dynamic delivery failed state contexts, defined in our defect model, accurately describe the existing dynamic delivery failures. To alleviate this threat, we explored multiple resources, including the Android documentation, issue repositories, discussion forums, and open-source Android apps' source code, to understand the root causes of dynamic delivery failures and identify contextual factors to induce them. Note that to construct a defect model for the purpose of test generation, we only need one situation in which the desired failed state is induced.

7 RELATED WORK

We provide an overview of the prior work on test-suite augmentation and test-input generation for mobile apps.

Test-Suite Augmentation: Software evolves, and so should the test suites to ensure validation of the evolving software. The purpose of test-suite evolution is two-fold [61, 74]. First, it can be used to repair available test cases for corrective regression testing [42, 45, 58, 72]. Second, developers can create additional tests for either progressive regression testing [34, 38, 39, 50, 63, 65, 67–71, 75–77] or to fulfill additional testing criteria, e.g., achieve higher coverage, find more crashes, or perform non-functional testing [31, 32, 40, 51, 59, 60, 62, 73]. Reuse of the available test suites to generate additional tests is known as *test augmentation*.

The closely related work to DeltaDroid are [31, 59, 62, 73]. Adamsen et al. proposed a technique that augments the existing test suites and systematically exposes them to adverse conditions where certain unexpected events may interfere with the test execution. They realized their technique in a tool, Thor, working on Android. Milani Fard et al. [59] proposed a technique that leverages existing test suites to automate the generation of tests for web applications. Specifically, they mine the human knowledge from the existing test suite to generate additional tests that explore uncovered parts of the program. Xuan et al. [73] proposed a technique to reproduce crashes by leveraging the existing test suites, instead of the automatic generation of new tests. Pradhan et al. [62] proposed a search-based technique to obtain the program dependence graph from the existing test suite to generate new tests to cover untested program configurations.

Although Thor [31] is similar in terms of using test augmentation techniques to provide contextual factors in running existing test suites, the events it injects into the test cases are substantially different from DeltaDroid's GUI and system events. It only generates events that might simulate two failed state contexts, namely *Network Error* and *Access Denied*. Therefore, it cannot be used to comprehensively test Android dynamic delivery failures. Furthermore, unlike [59, 62] which tries to achieve a higher *coverage of the code* by augmenting the existing test suites, the goal of DeltaDroid is to cover all the *Failed* states in the lifecycle of a DFM installation request through test augmentation. Moreover, While [73] relies on the information from a stack trace of a crash report to identify target tests and augments the existing test suite using random mutation operators, DeltaDroid relies on a novel defect model that formally defines failure contexts to generate effective tests.

Android Testing: Android test generation techniques mainly focus on either fuzzing to generate inputs or exercise an Android app through its GUI. The majority of recent techniques [37, 44, 66] rely on a GUI model, usually constructed dynamically and non-systematically, leading to unexplored program states. Sapienz [56], EvoDroid [55], and time-travel testing [41] employ an evolutionary algorithm. ACTEve [33], and Collider [49] utilize symbolic execution. CrawlDroid [35] introduces a feedback-based exploration strategy to effectively explore different states of Android apps. AimDroid [43] discovers unexplored activities with a reinforcement learning guided random algorithm. AppFlow [47] leverages machine learning to automatically recognize common screens and widgets and generate tests accordingly. MonkeyLab [53] mines app executions to generate GUI-based scenarios. Dynodroid [54] and Monkey [18] generate test inputs using random input values. Another group of techniques focuses on testing for specific defects [46, 48, 64, 78]. Mao et al. introduce a different approach [57] that generates replicable test scripts from crowd-based testing by collecting and analyzing test inputs from a crowd call to non-technical users with no specific software testing expertise or experience. None of these techniques can be used to properly validate the dynamic delivery in Android apps, as they cannot generate meaningful test inputs to induce the contextual settings for the manifestation of *Failed* states.

Similar to DFM-induced defects, energy defects occur under peculiar contextual settings. Jabbarvand et al. [48] proposed a search-based technique that leverages contextual models to generate both system and GUI test inputs. Unlike [48], which searches for contextual settings that manifest energy defects using meta-heuristics, we use a formal model of such contexts to generate tests for all of them.

8 CONCLUSION

In this article, we formally defined a novel defect model representing the conditions (contexts) under which the installation of a DFM in an Android app could fail. Utilizing the defect model, we introduced DeltaDroid, a test-suite augmentation approach for testing dynamic delivery in Android apps. DeltaDroid leverages static and dynamic analyses to detect the test cases that initiate the installation of DFMs in apps. DeltaDroid then modifies these tests to create new tests that augment the initial test suite to effectively create the conditions for reaching all of the *Failed* states. Our experimental results corroborate the effectiveness of DeltaDroid in inducing *Failed* states and detecting a significant number of installation defects among real-world Android apps. We have made DeltaDroid publicly available [20]. In our future work, we aim at extending DeltaDroid to test dynamic delivery in other platforms, e.g., Java Platform Module System [1].

REFERENCES

[1] 2017. Project Jigsaw. Retrieved 8 Dec. 2021 from http://openjdk.java.net/projects/jigsaw/.
[2] 2018. GitHub googlearchive Issues 1. Retrieved 8 Dec. 2021 from https://github.com/googlearchive/android-dynamic-features/issues/1.
[3] 2018. GitHub googlearchive Issues 2. Retrieved 8 Dec. 2021 from https://github.com/googlearchive/android-dynamic-features/issues/2.
[4] 2019. GitHub googlearchive Issues 36. Retrieved 8 Dec. 2021 from https://github.com/googlearchive/android-dynamic-features/issues/36.
[5] 2019. GitHub googlearchive Issues 41. Retrieved 8 Dec. 2021 from https://github.com/googlearchive/android-dynamic-features/issues/41.
[6] 2019. Network Exceptions in Android. Retrieved 8 Dec. 2021 from https://developer.android.com/guide/topics/connectivity/cronet/reference/org/chromium/net/NetworkException.
[7] 2019. Stackoverflow question 56255600. Retrieved 8 Dec. 2021 from https://stackoverflow.com/questions/56255600.
[8] 2019. Stackoverflow question 56454470. Retrieved 8 Dec. 2021 from https://stackoverflow.com/questions/56454470.
[9] 2019. Stackoverflow question 58611149. Retrieved 8 Dec. 2021 from https://stackoverflow.com/questions/58611149.
[10] 2020. About Android App Bundles. Retrieved 8 Dec. 2021 from https://developer.android.com/guide/app-bundle.
[11] 2020. Android Developers Blog: Recent Android App Bundle improvements and timeline for new apps on Google Play. Retrieved 8 Dec. 2021 from https://android-developers.googleblog.com/2020/08/recent-android-app-bundle-improvements.html.
[12] 2020. k-9 Mail App. Retrieved 8 Dec. 2021 from https://github.com/EverlastingHopeX/K-9-Modularization.
[13] 2020. Stackoverflow question 60181875. Retrieved 8 Dec. 2021 from https://stackoverflow.com/questions/60181875/.
[14] 2020. Stackoverflow question 60328502. Retrieved 8 Dec. 2021 from https://stackoverflow.com/questions/60328502.
[15] 2020. Stackoverflow question 61384372. Retrieved from https://stackoverflow.com/questions/61384372.
[16] 2020. Stackoverflow question 63100883. Retrieved 8 Dec. 2021 from https://stackoverflow.com/questions/63100883.
[17] 2020. Storage Exceptions in Android. Retrieved 8 Dec. 2021 from https://developers.google.com/android/reference/com/google/firebase/storage/StorageException.
[18] 2020. UI/Application Exerciser Monkey. Retrieved 8 Dec. 2021 from https://developer.android.com/studio/test/monkey.
[19] 2021. Appium Mobile App Testing Tool. Retrieved 8 Dec. 2021 from http://appium.io/.
[20] 2021. DeltaDroid. Retrieved 8 Dec. 2021 from https://sites.google.com/view/deltadroid/home.
[21] 2021. GitHub. Retrieved from https://github.com.
[22] 2021. Gradle Build Tool. Retrieved from https://gradle.org/.
[23] 2021. Play Core Library. Retrieved from https://developer.android.com/guide/playcore.
[24] 2021. Play Feature Delivery. Retrieved from https://developer.android.com/guide/playcore/dynamic-delivery.

[25] 2021. Play Feature Delivery - Handle request errors. Retrieved from https://developer.android.com/guide/playcore/feature-delivery/on-demand#handle_request_errors.
[26] 2021. StackOverFlow. Retrieved from https://stackoverflow.com.
[27] 2021. UX best practices for on demand delivery. Retrieved from https://developer.android.com/guide/playcore/feature-delivery/ux-guidelines#communicate.
[28] 2022. App Crawler. Retrieved from https://developer.android.com/studio/test/other-testing-tools/app-crawler.
[29] 2022. Thor Issue Repository. Retrieved from https://github.com/cs-au-dk/thor/issues/2.
[30] 2022. TimeMachine Issue Repository. Retrieved from https://github.com/DroidTest/TimeMachine/issues/7.
[31] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. 2015. Systematic execution of android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 83–93.
[32] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, and Nicola Amatucci. 2013. Considering context events in event-based testing of mobile applications. In *Proceedings of the 2013 IEEE 6th International Conference on Software Testing, Verification and Validation Workshops*. 126–133. DOI: https://doi.org/10.1109/ICSTW.2013.22
[33] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.
[34] Taweesup Apiwattanapong, Raul Santelices, Pavan Kumar Chittimalli, Alessandro Orso, and Mary Jean Harrold. 2006. Matrix: Maintenance-oriented testing requirements identifier and examiner. In *Proceedings of the Testing: Academic & Industrial Conference-Practice And Research Techniques*. IEEE, 137–146.
[35] Yuzhong Cao, Guoquan Wu, Wei Chen, and Jun Wei. 2018. Crawldroid: Effective model-based gui testing of android apps. In *Proceedings of the 10th Asia-Pacific Symposium on Internetware*. 1–6.
[36] S. R. Choudhary, A. Gorla, and A. Orso. 2015. Automated test input generation for android: Are we there yet? (E). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering*. 429–440.
[37] Faraz Yazdani Banafshe Daragh and Sam Malek. 2021. Deep GUI: Black-box GUI input generation with deep learning. In *Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 905–916.
[38] Thiago Botti de Assis, André Augusto Menegassi, and Andre Takeshi Endo. 2019. Amplifying tests for cross-platform apps through test patterns. In *Proceedings of the SEKE*. 55–74.
[39] Daniel Di Nardo, Fabrizio Pastore, and Lionel Briand. 2017. Augmenting field data for testing systems subject to incremental requirements changes. *ACM Transactions on Software Engineering and Methodology* 26, 1 (2017), 1–40.
[40] Zishuo Ding. 2019. *Towards the Use of the Readily Available Tests from the Release Pipeline as Performance Tests. Are We There Yet?* Ph. D. Dissertation. Concordia University.
[41] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of android apps. In *Proceedings of the 42nd International Conference on Software Engineering*. 1–12.
[42] Mark Grechanik, Qing Xie, and Chen Fu. 2009. Maintaining and evolving GUI-directed test scripts. In *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. IEEE, 408–418.
[43] Tianxiao Gu, Chun Cao, Tianchi Liu, Chengnian Sun, Jing Deng, Xiaoxing Ma, and Jian Lü. 2017. Aimdroid: Activity-insulated multi-level automated testing for android applications. In *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 103–114.
[44] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering*. IEEE, 269–280.
[45] Mark Harman and Nadia Alshahwan. 2008. Automated session data repair for web application regression testing. In *Proceedings of the 2008 1st International Conference on Software Testing, Verification, and Validation*. IEEE, 298–307.
[46] Roee Hay, Omer Tripp, and Marco Pistoia. 2015. Dynamic detection of inter-application communication vulnerabilities in Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 118–128.
[47] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: Using machine learning to synthesize robust, reusable UI tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 269–282.
[48] Reyhaneh Jabbarvand, Jun-Wei Lin, and Sam Malek. 2019. Search-based energy testing of Android. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering*. IEEE, 1119–1130.
[49] Casper S. Jensen, Mukul R. Prasad, and Anders Møller. 2013. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. 67–77.
[50] Yunho Kim, Zhihong Zu, Moonzoo Kim, Myra B. Cohen, and Gregg Rothermel. 2014. Hybrid directed test suite augmentation: An interleaving framework. In *Proceedings of the 2014 IEEE 7th International Conference on Software Testing, Verification and Validation*. IEEE, 263–272.
[51] Yavuz Koroglu and Alper Sen. 2018. TCM: Test case mutation to improve crash detection in Android. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*. Springer, 264–280.

[52] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The soot framework for Java program analysis: A retrospective. In *Proceedings of the Cetus Users and Compiler Infastructure Workshop*. 35.
[53] Mario Linares-Vásquez, Martin White, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2015. Mining android app usages for generating actionable gui-based execution scenarios. In *Proceedings of the 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 111–122.
[54] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 224–234.
[55] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 599–609.
[56] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 94–105.
[57] Ke Mao, Mark Harman, and Yue Jia. 2017. Crowd intelligence enhances automated mobile testing. In *Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 16–26.
[58] Atif M. Memon. 2008. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM Transactions on Software Engineering and Methodology* 18, 2 (2008), 1–36.
[59] Amin Milani Fard, Mehdi Mirzaaghaei, and Ali Mesbah. 2014. Leveraging existing tests in automated test generation for web applications. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. 67–78.
[60] Ana CR Paiva, André Restivo, and Sérgio Almeida. 2020. Test case generation based on mutations over user execution traces. *Software Quality Journal* 28, 3 (2020), 1173–1186.
[61] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. 2012. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.
[62] Dipesh Pradhan, Shuai Wang, Tao Yue, Shaukat Ali, and Marius Liaaen. 2019. Search-based test case implantation for testing untested configurations. *Information and Software Technology* 111 (2019), 22–36.
[63] Konstantin Rubinov. 2013. *Automatically Generating Complex Test Cases From Simple Ones*. Ph. D. Dissertation. Università della Svizzera italiana.
[64] Navid Salehnamadi, Abdulaziz Alshayban, Jun-Wei Lin, Iftekhar Ahmed, Stacy Branham, and Sam Malek. 2021. Latte: Use-case and assistive-service driven automated accessibility testing framework for android. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–11.
[65] Raul Santelices, Pavan Kumar Chittimalli, Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2008. Test-suite augmentation for evolving software. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 218–227.
[66] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 245–256.
[67] Haijun Wang, Xiaohong Guan, Qinghua Zheng, Ting Liu, Chao Shen, and Zijiang Yang. 2014. Directed test suite augmentation via exploiting program dependency. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*. 1–6.
[68] Zhihong Xu, Myra B. Cohen, Wayne Motycka, and Gregg Rothermel. 2013. Continuous test suite augmentation in software product lines. In *Proceedings of the 17th International Software Product Line Conference*. 52–61.
[69] Zhihong Xu, Yunho Kim, Moonzoo Kim, and Gregg Rothermel. 2011. A hybrid directed test suite augmentation technique. In *Proceedings of the 2011 IEEE 22nd International Symposium on Software Reliability Engineering*. IEEE, 150–159.
[70] Zhihong Xu, Yunho Kim, Moonzoo Kim, Gregg Rothermel, and Myra B. Cohen. 2010. Directed test suite augmentation: Techniques and tradeoffs. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 257–266.
[71] Zhihong Xu and Gregg Rothermel. 2009. Directed test suite augmentation. In *Proceedings of the 2009 16th Asia-Pacific Software Engineering Conference*. IEEE, 406–413.
[72] Jifeng Xuan, Benoit Cornu, Matias Martinez, Benoit Baudry, Lionel Seinturier, and Martin Monperrus. 2016. B-Refactoring: Automatic test code refactoring to improve dynamic analysis. *Information and Software Technology* 76 (2016), 65–80.
[73] Jifeng Xuan, Xiaoyuan Xie, and Martin Monperrus. 2015. Crash reproduction via test case mutation: Let existing test cases help. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 910–913.
[74] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability* 22, 2 (2012), 67–120.
[75] Tingting Yu. 2015. TACO: Test suite augmentation for concurrent programs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 918–921.

[76] Tingting Yu, Zunchen Huang, and Chao Wang. 2020. ConTesa: Directed test suite augmentation for concurrent software. *IEEE Transactions on Software Engineering* 46, 4 (2020), 405–419.
[77] Zhongxing Yu, Chenggang Bai, and Kai-Yuan Cai. 2013. Mutation-oriented test data augmentation for GUI software fault localization. *Information and Software Technology* 55, 12 (2013), 2076–2098.
[78] Li Lyna Zhang, Chieh-Jan Mike Liang, Yunxin Liu, and Enhong Chen. 2017. Systematically testing background services of mobile apps. In *Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 4–15.

Received 8 December 2021; revised 26 July 2022; accepted 29 July 2022