Toward In-Vivo Testing of Mobile Applications

Mariano Ceccato*, Luca Gazzola†, Fitsum Meshesha Kifetew*
Leonardo Mariani†, Matteo Orrú† and Paolo Tonella‡

*Fondazione Bruno Kessler (FBK), Trento, Italy
Email: {ceccato | kifetew}@fbk.eu
†*Università di Milano-Bicocca*, Milan, Italy
Email: {luca.gazzola | leonardo.mariani | matteo.orru}@unimib.it
‡*Università della Svizzera Italiana (USI)* Lugano, Switzerland
Email: paolo.tonella@usi.ch

***Abstract*—Mobile apps can be executed with an extremely large set of partially unpredictable configurations. Indeed, they can be executed on an unbounded combination of devices, operating systems, settings, and user preferences since apps may also interact with other apps or devices that were not even available when they were released. This results in a virtually infinite set of configurations that might be responsible for unexpected behaviors which can be validated in-house only to a negligible extent.**

To address this challenge, this paper discusses the application of in-vivo testing to mobile apps. The main idea is to run test cases in the field, where we exploit the intrinsic heterogeneity and variety of the end-user environment to dramatically increase the range of validated configurations. Actually, the many devices available in-the-field generate a naturally distributed and highly scalable environment that can be exploited to timely validate many configurations as soon as they are observed.

***Index Terms*—in-vivo testing, Android apps, configurations.**

I. Introduction

With the advent of mobile devices, more and more software applications are being offered as *mobile apps*, either exclusively or in parallel to their traditional variants (desktop and web applications). Indeed, mobile apps are nowadays part of a complex environment that consists of independent, though integrated and cooperating, systems of systems. As the mobile devices gain more computational and storage capacity, the level of complexity of the *apps* is also increasing. Such complexity manifests, among others, in terms of the number of configuration options that the *app* offers to the user. Such options have become quite commonplace, with strong support from the mobile platform producers. For instance, considering the Android platform, the most popular mobile operating system [25], developers have native support for managing the settings of their *apps* via the AndroidX Preference Library[1]. Unless there is a need for customized settings, developers can simply define the settings of their *app* and the library takes care of the rest (GUI, storage, saving/retrieving, etc.).

On the other hand, the version of the operating system itself, and the hardware and models of the mobile devices are also glowingly diverse. For the Android platform, there are 9 major versions currently active[2]. Furthermore, there is an increasing number of mobile device manufacturers, each producing and maintaining several models. A report found out that as of 2015 there were more than 24,000 distinct Android devices and over 1,000 device manufacturers [25]. The problems caused by such a multitude of devices and operating systems is known as the Android fragmentation problem [29]

Putting together the two spaces, that is, on one side the diversity of the devices and operating system versions, and on the other side the number of configurable options (settings/preferences) a specific *app* could have, the resulting configuration space is very large and it would be extremely time-consuming, and therefore unfeasible, for developers to test their apps on all these configurations before releasing them.

In addition, an app can interact with the other apps present in a device by emitting messages (*intents*). Since the association between an intent and the responding app is dynamic, apps that were not even available when the sending app was released could respond to intents, resulting in highly unpredictable scenarios. Consequently, many bugs may appear only after deployment and while an *app* is being used by the end users.

In-vivo testing, that is running the test cases in the field, directly in the devices of the end-users, can be an interesting option to address these challenges. Indeed, it can be used to exploit the heterogeneity of the devices, the operating systems, the preferences, and the app-specific settings of the real end-users to make the testing activity more effective. Although the configuration space to be explored can be very large, the configurations that matter, that is the ones used in practice, would be incrementally validated as soon as they appear in the field, with a potentially drastic increase of the effectiveness of the testing activity. In such a setting, the network of the end-user devices would constitute a very large testing platform that can be used to run many test cases in-vivo and to efficiently explore the configuration space.

However, deploying in-vivo testing on mobile devices is challenging, since it requires, for example, also deploying non-intrusive monitoring techniques to detect configurations and changes, isolation mechanisms, to prevent that any side-effect of the testing activity may impact on users, privacy preserving algorithms, to extract only non-sensitive information from the

[1]https://developer.android.com/guide/topics/ui/settings
[2]https://developer.android.com/about/dashboards/index.html

field.

In this paper, we discuss the challenges that we faced while working on in-vivo testing of mobile apps and we outline our proposed approach for in-vivo testing of Android apps. Although it is still a work-in-progress, our approach can already effectively handle the representation of the configuration space using feature models [14], the isolation of test cases using managed profiles[3], and the discovery of the configurations to be tested using monitoring mechanisms.

In the rest of the paper, we first discuss the challenges related to the definition of in-vivo testing for Android *apps* (Section II). We then discuss our approach to in-vivo testing (Section III). We compare our work to related work (Section IV), and finally present some concluding remarks (Section V).

II. Challenges

While designing our solution for enabling in-vivo testing of mobile apps, we identified a list of challenges that must be addressed to deliver an effective framework for in-vivo testing. In the rest of this section, we discuss these challenges.

Test space. An in-vivo testing framework must know both the features of the program that are *relevant* for in-vivo testing and the values that can be assigned to these features. For instance, the version of the operating system, the settings of the volume, the availability of the GPS, and the look and feel selected for a specific app are all examples of features that could be taken under consideration. The cartesian product of all feature-values defines the test space, that is, all the potential configurations in which the program could be executed. Since faults may manifest themselves only for specific configurations [19], the test space must be properly defined, without missing important features and without including irrelevant ones.

The challenge consists of capturing *all and only* the features that are relevant for testing, because missing some of them might result in some bugs being never revealed, while considering too many features (or too many feature values) might make the test space explode. An additional problem is how to represent this potentially very large test space in an efficient way, that could be embedded in apps with limited computational power.

Another challenge related to the test space is the management of its evolution. Indeed, the test space must evolve as soon as new features or new feature values turn out to be relevant, for instance when a new hardware is released in the market, when a new version of the OS is installed, and when the app under test is updated.

Configuration probing. To decide when to activate in-vivo testing, an app should be able to sense the configuration of the environment in which the app is deployed and is operating. However, this should be done in a way that is not invasive and annoying for the end-user, for instance without requiring additional permissions w.r.t. those that the end-user already grants to the app.

Tracking the status of the in-vivo testing process. The relevance of the configurations for in-vivo testing changes dynamically and a testing framework should be able to track this. For instance, an *untested* configuration might at some point be *tested*, thus becoming less interesting, or not interesting at all, for in-vivo testing. There might also be *unknown* feature values that at some point are discovered and thus become a target for in-vivo testing. In-vivo testing frameworks must be able to track the status of the tested features and configurations, and trigger the testing process when necessary.

An additional challenge for the framework is sharing this information across several devices. In fact, running in-vivo tests locally on an actual device may turn an *untested* configuration into a *tested* configuration. This change in the status should then be shared and propagated to all the other devices running the same app, so that no more resources are consumed for testing an already tested configuration.

Test execution. When a configuration must be tested, the framework has to retrieve the test cases and run them. The testing framework should take decisions about the test cases that can be executed *in-vivo* in the device [22] - e.g., the tests that can be well-supported by the isolation mechanisms available in the device - and the test cases that should be executed *ex-vivo* in-house [21] - e.g., the tests that can be isolated in-house only. Designing this logic might be challenging.

Moreover, the ex-vivo testing process may raise additional challenges related to the capability to replicate in-house a configuration that has been observed in the field. In particular, hardware configurations and configurations that include user sensitive information might be particularly hard to replicate.

Test case selection and generation. When testing is needed for a given configuration, the most appropriate tests should be identified and executed. The framework should execute the tests that better exercise the target configuration, skipping the irrelevant ones. This decision must take into account the specific configuration to be tested as well as the nature of the available test cases, which could be either unit, integration, or system test cases.

Test isolation. When test cases are executed in the end-user device, any *persistent side-effect must be prevented* (e.g., user data should not be deleted/modified by the tests cases) and *confidentiality of end-user data must be preserved* (e.g., user data should not be disclosed/exported by test cases). In vivo-testing must thus be equipped with proper sandboxing and privacy-preserving mechanisms that guarantee both properties. The challenge is designing mechanisms that provide the right level of guarantees. For instance, complete isolation of test execution would invalidate the benefit of in-vivo testing, whose goal is exploiting the end-user environment in the testing process. On the other hand, the environment must be exploited in a way that does not harm users and their data.

The actual tests could be manually written by the app developers, could be generated automatically, or could be a combination of the two. However, contrarily to regular unit,

[3]https://source.android.com/devices/tech/admin/managed-profiles

integration, and system tests executed in-house, the in-vivo test cases must support field execution. While some degree of isolation can be provided by the in-vivo framework, there might be other side-effects that the tests may generate and the framework cannot handle (e.g., due to the invocation of external services). Being either a human or a machine, the test generator should deal with the limitations of the adopted isolation solution.

Problem mitigation strategy. Although the primary goal of in-vivo testing is revealing faults and reporting them to developers, in-vivo frameworks have the opportunity to immediately improve the quality of the tested software by trying to automatically repair the tested app. For instance, the in-vivo framework may activate some *failure prevention/mitigation strategy* to fix the problem while waiting for a permanent fix released by the developers. Designing strategies and workarounds that can be actuated in a mobile device might be challenging, although some simple but effective policies can indeed be deployed, such as preventing the user from selecting a configuration that is known to cause app failures.

Performance cost. In order to be acceptable for the end users, an in-vivo testing framework must not impose any unacceptable degradation of the end-user experience or cost. The framework should be cheap in terms of memory requirements, network overhead, slow-down of the app execution, and energy consumption.

The cost of running the framework should be minimal either when the framework probes and monitors the current device configuration, but also when test cases are executed. For instance, running the tests should not cause a denial-of-service for the end-user (e.g., test cases should be executed only when the device is idle and not actively used).

The main performance challenge is represented by the contrasting goals of accurate testing with many test cases being executed, to increase the likelihood of detecting faults, and of cheap execution, because not too many resources should be consumed on the end-user device for the purpose of in-vivo testing.

III. Approach

In this section we introduce our in-vivo testing approach by presenting it on a small running example based on the ChatApp Android app.

A. ChatApp

Let us consider a hypothetical messaging app for Android devices, which we call *ChatApp* (pronounced shut-up). *ChatApp* supports the exchange of messages and multimedia content between its users. Moreover, *ChatApp* can take a picture of the user when the user creates or updates her profile. To take a picture, *ChatApp* sends an intent to delegate the task to any app that can take pictures using the camera of the mobile device (see Figure 1).

Since *ChatApp* relies on external resources (installed camera app; camera hardware) for the successful execution of the add/update profile image functionality, the scenarios in which a failure might happen depend on multiple factors: the hardware installed in the device, since the interaction with some camera models may fail; the configuration of the environment and operating system, since not all camera apps might be compatible with the ChatApp application; the settings of the app itself, since some specific choices might be not well supported by the app; and a combination of all these factors. Hence, adequately testing ChatApp requires addressing the combinatorial exploration resulting from all these factors.

```
1 Intent cameraIntent = new
      Intent(MediaStore.ACTION_IMAGE_CAPTURE);
2 cameraIntent.putExtra(MediaStore.EXTRA_OUTPUT,
      outputImgUri);
3 startActivityForResult(cameraIntent,
      REQUEST_IMAGE_CAPTURE);
```

Fig. 1. Intent sent by *ChatApp* to obtain a profile picture

B. Modelling the Configuration Space with Feature Models

To represent and manage the large configuration space that may affect apps, such as the ChatApp app, we use feature models [14]. Feature models provide a tree-like representation of the (combination of) features relevant to a product. In our case, the features represented in the model are the configurable items relevant to an app under test. We think feature models are a natural choice to effectively deal with such a large configuration space because they provide compact representations of highly combinatorial spaces.

The configuration model of *ChatApp* is shown in Figure 2, where inner nodes represent features; leaf nodes represent feature values; and the parent-child edges represent the feature-subfeature decomposition. While the default interpretation of feature decomposition is AND-decomposition, modifiers are available to express OR/XOR-decompositions and to identify a feature as mandatory/optional (see Legend in Figure 2). The logical constraints at the bottom-right are added to further constrain the admissible configurations.

The configuration of *ChatApp* is decomposed into two main parts: 1) *DeviceConfig*, representing the configuration of the device on which the app is running; and 2) *AppPrefs*, representing the various settings of the app itself. *DeviceConfig* includes the Android version (*OS* feature), the camera apps that can be delegated the task of taking a picture (*CameraApp*) and the actual model of the device (*DeviceModel*), all of which are mandatory features. In turn, *CameraApp* can be the default app (*Default*, mandatory feature) or an additional app (*Other*, optional feature). *Default* can be instantiated by a set of mutually exclusive apps (empty arc), while *Other* can be instantiated by a set of non exclusive apps (filled arc). When the device model is *CameraApp*, the camera hardware (*CameraHw* feature) can be either *IMX300* or *IMX400*.

ChatApp has also a couple of application-specific settings. The first one (*Upload*) represents a preference of the user to upload photos over wifi, mobile data, or both. The other setting (*Backup*) represents the preference of the user on whether

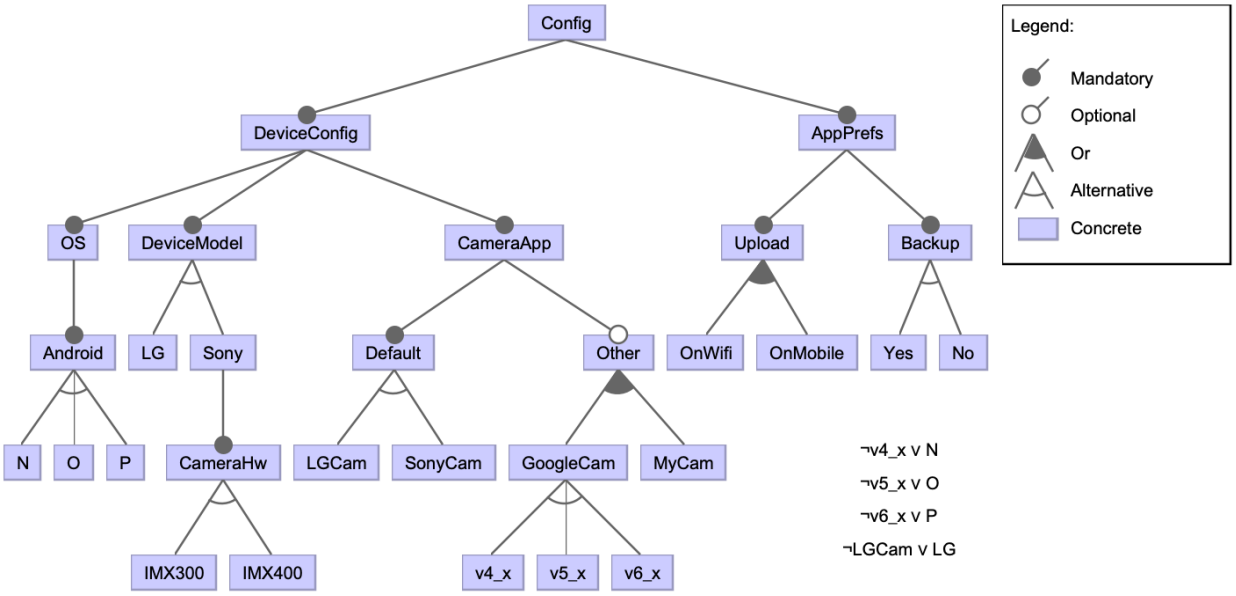

Fig. 2. Full configuration model for the *ChatApp* application

or not to backup chats. The feature model contains also a few cross-tree constraints of type "implies". For instance, the cross-tree constraint (v4_x $\Rightarrow$ N, equivalently shown as $\neg$4_x $\vee$ N in Figure 2) indicates that version 4_x of *GoogleCamera* constrains the version of *Android* to be N (*Nougat*); the camera app *SonyCamera* constrains the device model to be *Sony*.

Even in a small example such as the environment configuration model depicted in Figure 2, the total number of configurations admitted by the feature model is non trivial (288 valid feature configurations). Indeed, the feature model in Figure 2 is far from being a complete one: to keep its size manageable, only a small number of options have been expanded (just a few OS versions, device models, app preferences, etc). Testing all valid configurations exhaustively before deploying the app is not feasible because the number of combinations grows exponentially with the number of features and because some combinations might require very specific hardware/software components. Combinatorial testing (e.g., pairwise testing) [6] offers a way to systematically explore such a very large configuration spaces. However, by sampling a small representative fraction of all possible cases, it leaves several combinations untested. Some of them might be handled incorrectly by *ChatApp*, resulting in a runtime failure occurring in the field.

In addition to representing the full configuration space, we need to record also the set of configurations that have been tested so far. Let us consider *ChatApp* at the time it is first deployed to its users and let us assume that pre-release testing has been carried out on an LG phone with default camera on all three Android versions, with user settings specifying that upload is possible only on the wifi and that backup is disabled. The set of tested configurations will include the following tuples of feature values:

$\langle N, LG, LGCam, OnWifi, No \rangle$
$\langle O, LG, LGCam, OnWifi, No \rangle$
$\langle P, LG, LGCam, OnWifi, No \rangle$

The space of the configuration can grow to a significant size, as well as the the storage used to keep track of it. A very large size could also make search for already tested configurations more demanding in terms of time or computational resources. On the other hand, as the time goes by, some configuration can become obsolete (devices, operating system or app versions, just to name a few, can get out the market and be less and less adopted by the users). To this end, it would make sense, in the future, to foresee a "garbage collector" function, which could take care of automatically discarding the old configurations.

C. In-Vivo Testing of ChatApp

Our approach includes a run-time in-vivo test component that can monitor the configuration elements relevant to the app and checks whether the current configuration is:

- *tested* (i.e., exercised in pre-release testing): this means that the current configuration is valid according to the full configuration model and is among the tuples of tested configurations.
- *untested* (i.e., not exercised in pre-release testing): this means that the current configuration is valid, but it is not among the tuples of tested configurations.
- *unknown* (i.e., a configuration that includes a feature value not in the feature model): this means that the current configuration is not valid according to the full configuration model, which should be extended with a new leaf value for the newly discovered case.

This information can be extracted by a run-time probe that queries the device and the app preferences and compares the retrieved information to the tuples of tested configurations.

The following are examples of *tested*, *untested* and *unknown* configurations of *ChatApp*

tested ⟨*N, LG, LGCam, OnWifi, No*⟩
untested ⟨*N, Sony, SonyCamera, v4_x, IMX400, OnWifi, Yes*⟩
unknown ⟨*P, Xiaomi, XiaomiCamera, Xiaomi/Dual camera, v6_x, OnWifi, OnMobile, No*⟩

Different configurations trigger different reactions. A tested configuration triggers no reaction. An untested configuration, triggers in-vivo test execution. An unknown configuration triggers a feedback to testers who are asked to extend the model to incorporate the new cases that were not considered at the beginning, when the full configuration model was produced. In addition, an unknown configuration can be immediately validated with the available test cases.

When testing an app, the in-vivo framework must be able to select the relevant test cases and run them. This can be achieved with proper metadata that relate test cases to the exercised portion of the configuration space.

To guarantee isolation, we exploit a *managed profile*, a feature available in recent Android versions, designed to meet the *bring-your-own-device* policy, where employees can use their own private devices in a corporate-controlled environment. A managed profile represents an ideal technical solution for testing isolation, because it allows to separate the private user profile, where private user data are stored, from an ad-hoc (corporate) profile, where testing will take place. In this way, the side effects of testing will not affect user experience and user data.

Technically, an app installed in the managed profile is assigned a linux userid distinct from the one of the same app installed in the regular user profile, and the end-user sees two distinct (and well-marked) copies of the same app. These two apps will run as separate processes with their own distinct private memory and data space.

A managed profile is normally controlled by a *profile manager*. The profile manager can install and remove apps to/from the managed profile as needed, for instance only for the amount of time when testing is required. Additionally, the corporate profile manager can dynamically grant/revoke Android privileges to the apps installed in the managed profile (e.g., internet access, and the possibility to make phone calls), to implement the desired level of sandboxing and limit side effects. The possibility to monitor the apps that run in the managed profile (e.g., what URLs they access) is also a valuable support for testing.

If the executed test cases pass, the untested configurations become tested and the model of the untested configurations is updated accordingly. If a failure is detected, the failure is reported to the developer, and countermeasures to heal the execution and/or patch the program can be activated, if available.

Let us now consider the following hypothetical field failure:

> *A new camera app, XiaomiCamera, is installed. The camera hardware is deployed with a driver that, under Android version N, does not initialize the camera if not requested explicitly. When ChatApp takes a picture of the user, the request goes through XiaomiCamera, which does not explicitly initialize the camera when responding to an intent (it initialises the camera only when activated by the user). Correspondingly, XiaomiCamera crashes. In such a case, ChatApp times out the request to XiaomiCamera, leaving a reference to the requested picture set to null. When later the picture is used, a null pointer exception is thrown and ChatApp stops working.*

In such a scenario, the in-vivo testing component would:

1) recognize the configuration as unknown (in fact it does not appear in the feature model depicted in Figure 2);
2) execute the in-vivo tests, possibly retrieved from a testing server if not available locally, to check if *ChatApp* works properly with the camera app *XiaomiCamera*;
3) expose a failure of *ChatApp* (null pointer exception);
4) activate self-healing in the app (e.g., *ChatApp* may use an alternative or the default camera app).

In order to recognize the configuration with *XiaomiCamera* as unknown, our framework determines which apps can respond to the intent `MediaStore.ACTION_IMAGE_CAPTURE`. If any of such apps is not present under the pre-leaf node *Other*, a new leaf node is created and the configuration is marked as unknown. After finishing in-vivo testing, the in-vivo testing component communicates the new configuration and the results of testing to the server, which manages centrally all the configurations encountered and tested in the field, optionally triggering feedback to testers to incorporate a new case.

Recognition of a configuration as untested would trigger a similar reaction of the in-vivo testing component (check on the server, test execution, reporting to the server and possibly app healing). We envision the overall status of the field testing process, including the covered cases, the newly discovered configurations, and the available test cases to be monitored by testers with a dashboard.

IV. Related work

While to the best of our knowledge our work is the first to address in-vivo testing for mobile (specifically, Android) applications, there are previous works that deal with related problems. In particular, the problems of in-vivo monitoring and isolation have been already considered, though not in the mobile domain. Other related topics are ex-vivo testing using field data and empirical studies about field failures.

A. Techniques and approaches for in-vivo testing

In-vivo testing has been studied in recent years in several partially overlapping applicative domains, such as cloud computing [20], [31], distributed systems [24], web services [31], and embedded software [15], [24]. Existing approaches run the in-vivo test cases in the same environment where the software under test is operational, such that the testing infrastructure can share field resources with the tested product [1], [2], [28].

In-vivo testing has been used to address both functional and non-functional properties, often exploiting specification-based approaches based on choreographies and service-based specifications [1], [7], and finite-state models [5] to obtain the test cases. In the specific domain of mobile applications, in-vivo testing has received limited attention. Gu et al. [13] exploited in-vivo testing to validate the changes performed by repair actions, and Suliman et al. [27] proposed a framework for runtime testing of component-based applications using built-in tests. Differently from these works, this paper proposes to use in-vivo testing to address the very large configuration space that characterizes mobile applications, exploiting the end-user environment as a testbed that naturally offers the diversity required to effectively cover such a space.

B. Techniques to isolate in-vivo test execution

Several techniques [8], [11], [18], [22] have been proposed to support isolation during in-vivo testing. *Duplication* (also called *Cloning*) [8], [11], [22] consists of cloning the execution state (e.g., by forking a parallel process [22]) and executing in-vivo tests on the cloned execution state, hence ensuring that there is no interference with the end user execution of the application. Another proposed isolation mechanism is *Test mode execution* [3], [12], [15], [17], [24], [30], which allows to execute a component in a mode that does not interfere with regular executions (e.g., via data tagging). Transactional memory [4] can be used to perform in-vivo testing by creating a memory transaction whenever a test is executed. Then, such transaction is rolled back to prevent side-effects. Components can also be equipped with *built-in tests* [26] that are specifically designed for in-vivo test execution.

Existing approaches are hardly applicable in the context of Android because of limitations imposed by devices and operating systems. We thus addressed the isolation problem by taking advantage of the managed profiles available in the Android platform (see Section III).

C. Field Monitoring Techniques

Monitoring is a crucial activity for in-vivo testing. It is used to trigger test execution, to possibly start test generation and to eventually halt in-vivo testing. Monitoring may target different elements, including the target system [23], its events and states, and the system hosting the application under test [16]. The monitoring strategy closest to our approach is the one proposed by Lahami et al. [18], which monitors structural dynamic changes in the context of adaptable and distributed component-based systems. Their goal is to validate dynamically reconfigurable component-based systems upon change, so they monitor the runtime configuration of components and its changes, while we monitor the end user configuration of a mobile app.

D. Test generation using ex-vivo data

Ex-vivo testing consists of generating tests using data coming from production, with the purpose of running such tests in a testing environment which is isolated (or at least separated) from the production one.

In some cases, tests are created manually by the developers using ex-vivo data. For instance, Elbaum et al. [9] collected more than one thousand user sessions for the *Pine* email client, showing that field data have a strong ability to support test suite improvement.

Since ex-vivo testing cannot always reproduce untested/unknown configurations in-house, it has limited effectiveness when used with mobile applications. This justifies our proposal of in-vivo testing for such applications.

E. Empirical studies on field failures

Gazzola et al. [10] investigated the nature of field failures by analysing the bug reports of five applications. They introduce the notion of *field-intrinsic fault*, that is, a fault that is inherently hard to detect in-house. Their study shows that combinatorial explosion is the main cause of field-intrinsic faults. This finding motivates our proposal to move testing to the end user environment, where such combinatorial explosion can be dealt with thanks to the multitude of diverse installations and configurations available in the field.

V. Conclusion and future work

In-vivo testing can be promisingly used to exploit the resources available in the field to more effectively test software applications, taking advantage of the diversity and heterogeneity of the end-user environments and user-data. In this paper, we discussed the main challenges associated with designing in-vivo testing for mobile application, and Android applications in particular. We specifically described an approach that exploits feature models to manage the in-vivo testing process, configuration monitoring to trigger the field test process, and managed profiles to achieve isolation. This is an initial step towards the design of a complete distributed in-vivo testing solution for mobile apps.

We are now working toward the implementation and experimentation of our approach. Our future work includes advancing with the definition of cost-effective configuration probing strategies, designing test generation and execution strategies for in-vivo testing of mobile applications, and enriching the framework with policies to mitigate or repair problems that might be revealed during the in-vivo testing process.

Acknowledgements

This work has been partially supported by the Italian Ministry of Education, University, and Research (MIUR) with the PRIN project GAUSS (grant n. 2015KWREMX); by the H2020 Learn project, funded under the ERC Consolidator Grant 2014 program (ERC Grant Agreement n. 646867); by the H2020 Precrime project, funded under the ERC Advanced Grant 2017 program (ERC Grant Agreement n. 787703); by activities “API Assistant/STAnD” and “Teîchos” of the action lines *Digital Infrastructure* and *Digital Finance* of the EIT Digital.

We would like to thank Filip Ivanov Karchev for contributing to the implementation of the initial in-vivo prototype, in particular for engineering a solution for on-device execution of Espresso test cases and for contributing to the initial sketch of the in-vivo server.

REFERENCES

[1] M. Ali, F. De Angelis, D. Fan, A. Bertolino, G. De Angelis, and A. Polini. An extensible framework for online testing of choreographed services. *Computer*, 47(2):23–29, Feb 2014.

[2] X. Bai, S. Lee, W. Tsai, and Y. Chen. Ontology-based test modeling and partition testing of web services. In *2008 IEEE International Conference on Web Services*, pages 465–472, Sep. 2008.

[3] C. Bartolini, A. Bertolino, S. Elbaum, and E. Marchetti. Bringing white-box testing to service oriented architectures through a service oriented approach. *Journal of Systems and Software*, 84(4):655 – 668, 2011.

[4] J. Bobba, W. Xiong, L. Yen, M. D. Hill, and D. A. Wood. Stealthtest: Low overhead online software testing using transactional memory. In *18th International Conference on Parallel Architectures and Compilation Techniques*, pages 146–155, Sep. 2009.

[5] D. Brenner, C. Atkinson, B. Paech, R. Malaka, M. Merdes, and D. Suliman. Reducing verification effort in component-based software engineering through built-in testing. In *2006 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06)*, pages 175–184, Oct 2006.

[6] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. Constructing test suites for interaction testing. In *Proceedings of the 25th International Conference on Software Engineering ICSE*, pages 38–48, 2003.

[7] M. B. Cooray, J. H. Hamlyn-Harris, and R. G. Merkel. Dynamic test reconfigurationfor composite web services. *IEEE Transactions on Services Computing*, 8(4):576–585, July 2015.

[8] H. Dai, C. Murphy, and G. E. Kaiser. CONFU: configuration fuzzing testing framework for software vulnerability detection. *International Journal of System of Systems Engineering*, 1(3):41–55, 2010.

[9] S. Elbaum and M. Hardojo. An empirical study of profiling strategies for released software and their impact on testing activities. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 65–75, New York, NY, USA, 2004. ACM.

[10] L. Gazzola, L. Mariani, F. Pastore, and M. Pezzè. An exploratory study of field failures. In *28th IEEE International Symposium on Software Reliability Engineering, ISSRE*, pages 67–77, 2017.

[11] A. González-Sanchez, É. Piel, and H. Groß. Ritmo: A method for runtime testability measurement and optimisation. In *Proceedings of the Ninth International Conference on Quality Software, QSIC*, pages 377–382, 2009.

[12] M. Greiler, H.-G. Gross, and A. van Deursen. Evaluation of online testing for services: a case study. In *Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems*, pages 36–42. ACM, 2010.

[13] T. Gu, C. Sun, X. Ma, J. L, and Z. Su. Automatic runtime recovery via error handler synthesis. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 684–695, Sep. 2016.

[14] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon University – Software Engineering Institute, November 1990.

[15] K. Kawano, M. Orimo, and K. Mori. Autonomous decentralized system test technique. In *[1989] Proceedings of the Thirteenth Annual International Computer Software Applications Conference*, pages 52–57, Sep. 1989.

[16] T. M. King, D. Babich, J. Alava, P. J. Clarke, and R. Stevens. Towards self-testing in autonomic computing systems. In *Proceedings of the International Symposium on Autonomous Decentralized Systems*, ISADS '07, pages 51–58, 2007.

[17] M. Lahami, M. Krichen, and M. Jmaiel. Safe and efficient runtime testing framework applied in dynamic and distributed systems. *Science of Computer Programming*, 122:1–28, 2016.

[18] M. Lahami, M. Krichen, and M. Jmael. Runtime Testing Approach of Structural Adaptations for Dynamic and Distributed Systems. *Int. J. Comput. Appl. Technol.*, 51(4):259–272, July 2015.

[19] Y. Lu, M. Pan, J. Zhai, T. Zhang, and X. Li. Preference-wise testing for android applications. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019.

[20] A. Metzger, O. Sammodi, K. Pohl, and M. Rzepka. Towards pro-active adaptation with confidence: Augmenting service monitoring with online testing. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 20–28, New York, NY, USA, 2010. ACM.

[21] J. Morn, A. Bertolino, C. de la Riva, and J. Tuya. Towards ex vivo testing of mapreduce applications. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 73–80, July 2017.

[22] C. Murphy, G. Kaiser, I. Vo, and M. Chu. Quality Assurance of Software Applications Using the In Vivo Testing Approach. In *2009 International Conference on Software Testing Verification and Validation*, pages 111–120, Apr. 2009.

[23] C. Murphy, M. Vaughan, W. Ilahi, and G. Kaiser. Automatic detection of previously-unseen application states for deployment environment testing and analysis. In *Proceedings of the 5th Workshop on Automation of Software Test*, pages 16–23. ACM, 2010.

[24] E. Nishijima, H. Yamamoto, K. Kawano, K. Fujiwara, and K. Oshima. On-line testing for application software of widely distributed system. In *Proceedings 15th Symposium on Reliable Distributed Systems*, pages 54–63, Oct 1996.

[25] Opensignal. Android fragmentation visualized. https://www.opensignal.com/sites/opensignal-com/files/data/reports/global/data-2015-08/2015_08_fragmentation_report.pdf. Accessed 4 September 2019.

[26] O. Sammodi, A. Metzger, X. Franch, M. Oriol, J. Marco, and K. Pohl. Usage-Based Online Testing for Proactive Adaptation of Service-Based Applications. In *2011 IEEE 35th Annual Computer Software and Applications Conference*, pages 582–587, 2011.

[27] D. Suliman, B. Paech, L. Borner, C. Atkinson, D. Brenner, M. Merdes, and R. Malaka. The morabit approach to runtime component testing. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, volume 2, pages 171–176, Sep. 2006.

[28] Y. Wang, X. Bai, J. Li, and R. Huang. Ontology-based test case generation for testing web services. In *Eighth International Symposium on Autonomous Decentralized Systems (ISADS'07)*, pages 43–50, March 2007.

[29] L. Wei, Y. Liu, and S.-C. Cheung. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016.

[30] H. Zhu and Y. Zhang. Collaborative testing of web services. *IEEE Transactions on Services Computing*, 5:116–130, 2012.

[31] H. Zhu and Y. Zhang. *A Test Automation Framework for Collaborative Testing of Web Service Dynamic Compositions*, pages 171–197. Springer New York, New York, NY, 2014.