DroidMate-2: A Platform for Android Test Generation

Nataniel P. Borges Jr.*
Saarland University
Saarbrücken, Germany
nataniel.borges@cispa.saarland

Jenny Hotzkow
Saarland University
Saarbrücken, Germany
jenny.hotzkow@cispa.saarland

Andreas Zeller
Saarland University
Saarbrücken, Germany
zeller@cs.uni-saarland.de

ABSTRACT

Android applications (*apps*) represent an ever increasing portion of the software market. Automated test input generators are the state of the art for testing and security analysis.

We introduce DroidMate-2 (DM-2), a platform to easily assist both developers and researchers to customize, develop and test new test generators. DM-2 can be used without app instrumentation or operating system modifications, as a test generator on real devices and emulators for app testing or regression testing. Additionally, it provides sensitive resource monitoring or blocking capabilities through a lightweight app instrumentation, out-of-the-box statement coverage measurement through a fully-fledged app instrumentation and native experiment reproducibility. In our experiments we compared DM-2 against DroidBot, a state-of-the-art test generator by measuring statement coverage. Our results show that DM-2 reached 96% of its peak coverage in less than 2/3 of the time needed by DroidBot, allowing for better and more efficient tests. On short runs (5 minutes) DM-2 outperformed DroidBot by *7%* while in longer runs (1 hour) this difference increases to *8%*.
ACM DL Artifact: https://www.doi.org/10.1145/3264864. For the details see:

https://github.com/uds-se/droidmate/wiki/ASE-2018:-Data

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Dynamic analysis*; • **Human-centered computing →** Graphical user interfaces; Smartphones;

KEYWORDS

dynamic analysis, test generation, Android

ACM Reference Format:
Nataniel P. Borges Jr., Jenny Hotzkow, and Andreas Zeller. 2018. DroidMate-2: A Platform for Android Test Generation. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3238147.3240479

*Authors in alphabetical order.

This work is licensed under a Creative Commons Attribution International 4.0 License.

ASE '18, September 3–7, 2018, Montpellier, France

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3240479

1 INTRODUCTION

The Android mobile application market is highly volatile and competitive with millions of applications (*apps*) on Google Play Store. The quality of an app is vital to stay competitive in such a market and testing is a core technique for quality control.

Since manual app testing is expensive and laborious, several tools for automated test generation [10] have been developed. Automated testing is, however, a constantly evolving subject, as testing techniques improve, so increases the app's complexity. This results in a never ending demand for more advanced testing techniques to cover app behavior as effective and efficient as possible. Quality control mostly implies defect-free apps and test generators mainly focus on functionality testing. As the recent Facebook incident [5] shows, this is no longer sufficient. As security and privacy become more prominent, security analysis should be part of the testing and development cycle.

In this work we present DM-2, an extended and improved version of the original DroidMate project. While DroidMate was a test input generator with API monitoring capabilities, DM-2 offers easy to use mechanics to implement systematic testing strategies on top of a ready to use selection of strategies such as random, fitness based or playback of recorded executions. DM-2 is still usable out-of-the-box as a random test input generator, however, besides major performance improvements, its improved design gives developers and researchers means to easily implement and combine their own custom strategies while abstracting all Android specifics, such as app setup or device communication, away. Each strategy is automatically selected according to freely configurable conditions, e.g., the condition *if there is a permission request, accept it* can be easily expressed as boolean check of the currently visible elements for the permission request identifier. All these test generation algorithms can benefit from monitoring, mocking or blocking of sensitive resources during analysis as well of a new dynamically generated and extensible app model, which allows for context aware exploration strategies. DM-2 runs on stock Android versions between 6 (API 23) and 8 (API 26), on physical devices and emulators without any need of device rooting or operating system (OS) modification. It provides out-of-the-box test reproducibility, as well as statement coverage analysis via app instrumentation.

2 TOOL DESIGN

The architecture of DM-2 consists of three major components, as illustrated by the dashed frames in Figure 1.

Exploration Engine – running on the host PC – creates the UI state model (on the fly) and determines interactions with the app under test based on a configurable set of exploration strategies;
Monitoring Proxy – running on the device – is responsible for intercepting API calls between the app and the Android OS to log, block or mock the responses of these API calls;

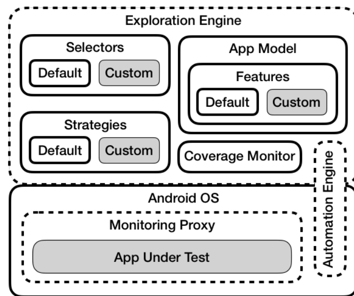

Figure 1: DM-2's conceptual architecture, highlighting places where the developer can add custom implementations (light gray), as well as components from the original DroidMate extended by DM-2 (dashed).

Automation Engine – consisting of a PC interface and a device component which communicate via TCP – to execute the actions as determined by the *Exploration Engine* on the device. Thereby the device itself is controlled via Android's native UiAutomator.

2.1 Exploration Engine

Originally, the test generation of DroidMate utilized an *exploration loop* to determine after each executed device action what interactions should be issued next. DM-2 extends this mechanism with a pool of exploration *strategies* and *selectors* which determine the next interaction, based on the current state of the *App Model* (see Section 2.1.3). Each *selector* programmatically specifies, based on the current and previous model states, if it is able to process the current state and, if so, which specific strategy should be activated. To cope with cases where multiple conditions are fulfilled at the same time, a unique priority value has to be specified for each selector. For any iteration in the exploration loop, the selector with the highest priority which can process the current state is chosen and its strategy is used to compute the next device interaction.

After each device action DM-2 fetches the current device screen (window dump) and a screenshot, alongside potential logcat exceptions. With this information the new app state is derived as described in *App Model*. Additionally, all registered *model features* are notified about this state transition. These model features provide the interface to allow for model extensions like a blacklist of all UI elements which let the app crash or the exploration stagnate.

The new state is then used in the next iteration of the exploration loop as *current state* of the app. This loop continues until any strategy triggers a terminate action.

2.1.1 Strategy Selectors. DM-2 supports the definition of criteria to determine the best strategy for different situations, e.g., when a permission is requested or the exploration gets stuck. These criteria are referred to as *selectors*.

Formally, a selector is defined as a pair $\left(p, f(c) \rightarrow s\right)$ where p is its priority and $f(c)$ is a mapping function f from a model context c (e.g., the current state or the action trace) to an exploration strategy s. DM-2 calculates if the selection criteria is fulfilled $f(c)$ and choses the most significant successful selector. That is, the one with highest priority which returned a strategy s to be used to derive the next device interaction.

DM-2 provides a set of selectors to activate the default strategies described in Section 2.1.2. The set of selectors to be used can be configured via command line or within a configuration file. Custom selectors can be implemented or existing selectors can be modified (e.g to change priorities) and passed to DM-2's initialization.

2.1.2 Strategies. An interaction with an app can be simple, such as *click on coordinates* (x, y), or complex, such as *close the app, enable the device wi-fi, bluetooth and restart the app from its initial screen.* Such complex interactions are abstracted as *exploration actions*. An exploration action determines which specific sequence of *device actions* should be performed by the automation engine. A strategy decides, based on the current state of the app model, which exploration action(s) should be issued next.

DM-2 is shipped with a set of strategies, e.g., it is able to randomly explore an arbitrary app without any need of additional meta information, labeling or manual user interaction. In addition to random exploration, DM-2 offers the following default strategies

Reset (re-)enables wi-fi, triggers the home button and starts the app from its main activity.
Terminate closes the app and finishes the exploration.
Back presses the back button of the device.
BiasedRandom randomly selects a UI element from the current screen, among those which have been least explored, and clicks or long clicks it.
In particular, we count how often any UI element was clicked in the context of a specific state and how often over all states. The least interacted UI element is computed by determining the interactable elements which were least interacted in the context of the current state and filtering these by the smallest overall interaction number. If this computation results in more than one UI element, the target is chosen randomly among them. Apps which belong to the application package are preferred to guide the exploration rather to app internal features.
Random randomly clicks on the coordinate of any UI element on the screen, similar to Monkey [1].
Fitness Proportionate uses a statically mined interaction model to predict the probability of each UI element having an event, then use these probabilities as bias for a random selection.
PlayBack selects the next valid interaction from a previously recorded model trace and replays it.

2.1.3 App Model. The app model constructed during exploration consists of the UI *states* – featuring a set of UI elements. Some of these UI elements are *interactable*, meaning the user can, for example, tick, click or long-click them. This interaction (*transition*) may lead into another state. While Androids allows the developer to specify a *resource id* for any UI element, it is optional and seldom used. Thus, we uniquely identify a UI element by computing its id *wId* in the form of a *UUID* (from the Java default library) based on the concatenation of its display and description text, if available, or on its image bytes, cut from a screenshot, if not. In addition to the unique ID, we compute a *propId* which represents the UI element configuration by converting all state-related properties such as position, checked, enabled, click-able etc. to a string and computing its UUID.

We aim to identify conceptually identical states, i.e., slight differences in the rendering like the highlighting of previously interacted elements should not be interpreted as a different state. Moreover, we disregard UI elements which do not belong to the app, that is, possess a different package name; as well as non-interactable and simply structuring elements as non-essential for the conceptual identity of the UI state. However, we would still like to be able to distinguish if, for example, a check-box in the same conceptual state was ticked or not. With this goal, we specify the identity *id* of a state by a tuple $stateId = (uniqueId, configId)$. Where the *uniqueId* is computed as the union of the *wId* of all UI elements belonging to the app which are either interactable or leafs in the element hierarchy (non-leaf elements are used for layout and visual representation purposes) and the *configId* is the union of the *propId* of all currently visible UI elements.

This metric for unique ids allows us to efficiently re-identify conceptually identical UI states as well as UI elements which reoccur within different UI states (like menu or help buttons), independently of their position or layout. The metric for *configId* allows to identify which configurations were explored for each UI state. Both functionalities are essential for advanced exploration strategies.

2.2 Automation Engine

The *Automation Engine* abstracts and manages all communication between the exploration and app using a synchronous protocol based on *actions* and *responses*. Strategies send one action at a time to the *Automation Engine* (PC side), which forwards it to its on-device instance and halts the exploration while this action is processed.

The on-device *Automation Engine* converts actions into automation commands or API calls. Before issuing a response to the exploration, the on-device *Automation Engine* must wait until the app stabilizes, that is, until it finishes performing the previous action and has at least one element from the app to interact with. This synchronization is a natural bottleneck of most Android test generators which allow for state-aware UI actions. It is, however, necessary to correctly emulate a user's behavior, who would have to wait until a new screen is loaded to continue the exploration.

The time to execute an action varies according to the functionality being performed – ticking a checkbox is faster than clicking a login button – as well as external factors, such as network speed and server availability. The *Automation Engine* copes with varying load times as follows: first, it waits for the device to be idle, that is, ready to receive and handle commands again. It then waits until at least one UI element can be interacted with. It discards any UI elements displayed only during this transition period, e.g., progress bars, as they do not provide any explorable behavior.

Once the app stabilizes, the on-device *Automation Engine* issues a response containing the structural (screen dump) and visual (screenshot) state of the device to its PC counterpart, which forwards it to the *Exploration Engine* for processing.

2.3 Monitoring Proxy

The *Monitoring Proxy* is a payload deployed to the device in order to work as a proxy between the app and OS. It monitors a configurable list of privacy-sensitive resources and intercepts API invocations without changing the app code. This functionality is only available on physical devices or emulators with ARM processor architecture. In order to use it, the app has to be instrumented to activate this payload before starting the app. DM-2 offers this instrumentation functionality through its *inline* mechanism, which inserts an activation call to the payload in the entry point of the app, keeping the remainder of the app unaltered.

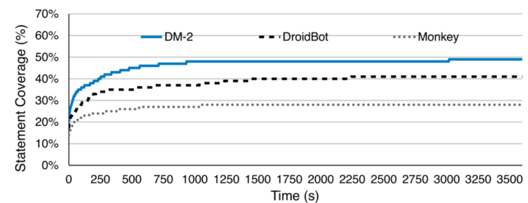

Figure 2: Average coverage over time between DM-2, DroidBot and Monkey for the test dataset.

While originally developed for API monitoring, the *Monitoring Proxy* allows custom code to be triggered for each API, as well as to individually define security policies such as mocking or blocking API access. When active, *Monitoring Proxy*'s standard mocking behavior is to return a default value for primitive type APIs, such as 0 for integers and empty string for strings. When blocking, it by default raises a security exception.

3 EMPIRICAL EVALUATION

We conducted a set of experiments to evaluate DM-2's efficiency and effectiveness. In particular we aim to answer the following research question: *Does DM-2 reach a coverage peak faster than current tools?*

We compared DM-2 against DroidBot, which presents similar functionality and has been shown to outperform most current test generators [2] and Monkey, the standard test generator from Android. As an evaluation metric we used statement coverage, which has been extensively used to determine the effectiveness of testing tools and is regarded as a good predictor for fault detection [6]. We evaluated all tools on 11 different apps, randomly chosen from [3]. Each app exploration was executed for 1 hour on a real Google Nexus 5X device, with 10 runs per app to mitigate noise.

Our results in Figure 2 show that concrete strategies are superior to pure random events for the tested app set. Even though Monkey implements more input types (e.g., swipe and zoom) than the other tools it has the worst overall coverage. This is in particular true for apps which have 'deeper functionality', meaning the user has to navigate through a few screens until certain features are accessible.

DroidBot and DM-2, both, try to systematically explore different UI elements. DroidBot applies a depth-first strategy, meanwhile DM-2 uses *BiasedRandom*. This fact together with the better performance of DM-2 (2s instead of 3-4s per action) lead to faster and more efficient explorations. On average, DM-2 achieved 8% better coverage than DroidBot after 1 hour, with a 7% difference in 5 minute runs. In addition, this chart shows that DM-2 achieved 96% of its maximum coverage in approximately 15 minutes, while DroidBot reached the same ration in approximately 24 minutes.

4 USAGE SCENARIOS

We envision DM-2 as a tool beneficial to both research and industry due to its modular architecture and out-of-the-box functionality. Below we describe a few scenarios where DM-2 can be applied.

Record and Replay System Tests. During the test execution, DM-2 records each action performed, UI element seen and state reached, it also provides out-of-the-box statement coverage measurement. An initial DM-2 run – or a set of runs with different configurations – can be executed to test a system. During the release of a new version of the app these tests can be replayed and the following evaluations performed: (1) *does the functionality tested on the previous app version still work the same?* (2) *Are there any previously tested code segments which are no longer tested?*

Sensitive Resource Impact Analysis. DM-2 can be used to analyze the impact of access restriction to sensitive resources. While DroidMate was capable of monitoring API access it did not provide any means to restrict – or impersonate – them. By combining DM-2's API restricting capabilities with its record and replay feature and code coverage metric, it is possible to measure the impact when restricting access to a sensitive resource.

App Behavior Analysis. DM-2 creates an app model during exploration, which can be used to generate new test inputs. This model can be easily extended with custom features, like tracking APIs triggered after clicking UI elements with certain text labels.

Previous research [7] demonstrated that such information can be used to identify anomalous behaviors within app categories. Our model allows such studies to be performed with context-awareness (current and previous app states) and on finer granularity level.

New Test Generation Strategies. DM-2's architecture offers major assistance for the development of new Android test generation algorithms. DM-2 can be used to abstract all low-level Android communication and to create tests from a model representation of the app, mitigating the development efforts. In addition, DM-2 provides natively experiment reproducibility (through record and replay) as well as coverage metrics for comparison between techniques.

5 RELATED WORK

Automated Test Input Generation in mobile apps is a frequent research topic. Three major approaches are used for input generators, namely: *random*, *model-based*, and *systematic*.

Random testing tools create sequences of events to explore apps. Representatives are Monkey [1] – the lightweight test generator shipped with Android, as well as DroidMate [8]. DM-2 supports the same feature, while providing more functionality.

Model-based input generators use a – previously defined or on the fly generated – model to produce inputs. For example DroidBot [9] employs different methods to dynamically construct and consume app models. DM-2's offers similar features. A developer can easily implement custom strategies which use the on the fly constructed model. DroidBot's methods to quantify test effectiveness are orthogonal to our approach and may be integrated in the future. *SwiftHand* [4] uses statically generated models to guide test generation. DM-2 provides out-of-the-box a *Fitness Proportionate* strategy which consumes a static model for test generation. This mechanism can be applied to arbitrary input models.

Systematic testing approaches systematically test an app with a specific goal. *EvoDroid* [11] and *Sapienz* [12] attempt to improve test coverage, while *IntelliDroid* [13] attempt to trigger specific behaviors. With DM-2's extensive app model it is straightforward to support all these approaches or to try out other new strategies.

Besides the performance benefits as presented in Section 3, DM-2 offers additional features like reproducibility (playback strategy), an extensible architecture and a more powerful app model.

6 CONCLUSION

We present DM-2, a platform for Android test generation, which can be used on all stock Android versions 6 to 8.1. While DM-2 can be used out of the box for random testing, with API monitoring and privilege restriction, or for recording and replaying system tests, its major advantage is the easement for development, combination, extension and comparison of different test generation strategies. DM-2 simplifies the work for developers by abstracting all device related issues and providing a dynamically constructed app model, which can be used to verify test criteria or for strategy development. Our evaluation showed that DM-2 is able not only to achieve a better coverage than both a state-of-the-art and Android's standard test generators, but that it is also able to reach its peak coverage earlier, allowing for faster more efficient testing.

ACKNOWLEDGMENT

This work was funded partially by a German Research Foundation (DFG) grant (D514111409), the DFG project (SFB 1223, Project A3) and partially by European Research Council grant (G514111401).

REFERENCES

[1] Android. 2017. UI/Application Exerciser Monkey. (2017). https://developer.android.com/studio/test/monkey.html
[2] Lingfeng Bao, Tien-Duy B Le, and David Lo. 2018. Mining sandboxes: Are we there yet?. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 445–455.
[3] Nataniel Borges Jr., Maria Gómez, and Andreas Zeller. 2018. Guiding App Testing With Mined Interaction Models. In *Mobile Software Engineering and Systems (MOBILESoft'18), 2018 IEEE/ACM International Conference on*. IEEE.
[4] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *OOPSLA '13 Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 1–30.
[5] Facebook. 2017. Cracking Down on Platform Abuse. (2017). https://newsroom.fb.com/news/2018/03/cracking-down-on-platform-abuse/
[6] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 72–82.
[7] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. 2014. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 1025–1035.
[8] Konrad Jamrozik and Philipp Von Styp-rekowsky Andreas. 2016. Mining Sandboxes. *ICSE, 2016 IEEE/ACM 38th International Conference on Software Engineering* (2016).
[9] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot : A Lightweight UI-Guided Test Input Generator for Android. *2017 IEEE/ACM 39th IEEE International Conference on Software Engineering* (2017).
[10] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. 2017. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 399–410.
[11] 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 – FSE'14*. 599–609.
[12] 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 (ISSTA 2016)*. ACM, New York, NY, USA, 94–105.
[13] Michelle Y Wong and David Lie. 2016. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 21–24.