A Framework for Automated Test Mocking of Mobile Apps

Mattia Fazzini
University of Minnesota
Minneapolis, MN, USA
mfazzini@umn.edu

Alessandra Gorla
IMDEA Software Institute
Madrid, Spain
alessandra.gorla@imdea.org

Alessandro Orso
Georgia Institute of Technology
Atlanta, GA, USA
orso@cc.gatech.edu

ABSTRACT

Mobile apps interact with their environment extensively, and these interactions can complicate testing activities because test cases may need a complete environment to be executed. Interactions with the environment can also introduce test flakiness, for instance when the environment behaves in non-deterministic ways. For these reasons, it is common to create test mocks that can eliminate the need for (part of) the environment to be present during testing. Manual mock creation, however, can be extremely time consuming and error-prone. Moreover, the generated mocks can typically only be used in the context of the specific tests for which they were created. To address these issues, we propose MOKA, a general framework for collecting and generating reusable test mocks in an automated way. MOKA leverages the ability to observe a large number of interactions between an application and its environment and uses an iterative approach to generate two possible, alternative types of mocks with different reusability characteristics: advanced mocks generated through program synthesis (ideally) and basic record-replay-based mocks (as a fallback solution). In this paper, we describe the new ideas behind MOKA, its main characteristics, a preliminary empirical study, and a set of possible applications that could benefit from our framework.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Test mocking, mobile apps, software environment

ACM Reference Format:
Mattia Fazzini, Alessandra Gorla, and Alessandro Orso. 2020. A Framework for Automated Test Mocking of Mobile Apps. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3324884.3418927

1 INTRODUCTION

Nowadays, we use mobile applications (or simply apps) for many of our daily activities, including reading the news, streaming content, and communicating with friends and family. Because some of these apps are used daily by millions of users, it is fundamental to thoroughly assess their quality and avoid failures due to undetected bugs. Although testing has been shown to be effective in identifying bugs, these apps typically have a large number of interactions with their software environment (e.g., their underlying application framework) that can affect and complicate testing activities [1].

Specifically, tests that involve interactions with the environment might (1) suffer from flakiness [2, 3], need a complete (and often complex to set up) execution environment, and/or require a long time to complete. A common strategy to mitigate this kind of issues during testing is to manually create test mocks[1] for specific test executions. Unfortunately, manual mock creation is typically a time-consuming and a error-prone task. Moreover, usually these mocks can not be reused across tests—let alone apps—as they are specifically crafted for the specific context they target.

To address these issues with the creation and use of test mocks, we propose a framework called MOKA. Our framework leverages existing tests to generate general and reusable test mocks that can also be used with new tests. More precisely, given an app under test (AUT) and its test suite, MOKA collects mock data from test executions, uses program synthesis to generate test mocks from the data, and refines the mocks by repeating the synthesis task while incrementally considering new mock data collected from test executions of other apps and through input generation. Our framework collects mock data at the AUT's *interaction points*—code entities used by the app to interact with external code (e.g., a method of the application framework called by the app).

Although the problem of generating test mocks has been investigated before [6–9], to the best of our knowledge MOKA provides the first comprehensive framework for generating test mocks across mobile apps, and makes a leap forward toward having reusable test mocks. In this paper, we also present a preliminary empirical study based on 20 apps that highlights MOKA's potential. In the study, we analyzed more than 4000 tests and found that 30% of the developers' defined mocks target the application framework, thus providing evidence of the potential usefulness of our proposed approach. Finally, the paper discusses how MOKA's test mocks can be useful for automated input generation, test evolution, and cloud-based testing. This paper makes the following contributions:

- The description of MOKA, our proposed framework for generating reusable test mocks.
- A preliminary empirical study based on 20 apps that highlights the framework's potential.
- A discussion of how MOKA's test mocks could be helpful in multiple automated-testing scenarios.

[1] Although the terms *test mock*, *test stub*, and *test fake* are used to indicate slightly different concepts [4, 5], in this paper we only use the term *test mock* for simplicity.

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3418927

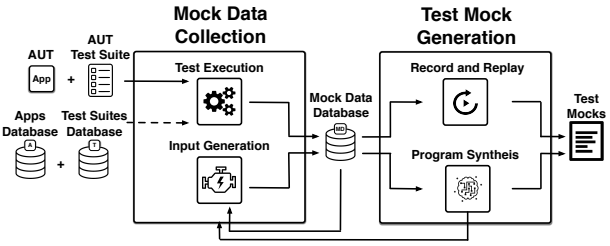

Figure 1: High-level overview of the MOKA framework.

2 THE MOKA FRAMEWORK

This section presents MOKA, our envisioned framework for generating reusable test mocks (*RTMs*). Figure 1 provides a high-level overview of MOKA. As the figure shows, the framework takes as inputs an AUT and a set of tests for the AUT and produces RTMs as its output. MOKA iterates through two main phases: *mock data collection* and *test mock generation*. The mock data collection phase starts by running the tests (provided as input) on the AUT and collects mock data. The test mock generation phase creates RTMs through program synthesis based on available mock data. In the test mock generation process, the framework uses an iterative approach that incrementally considered new mock data gathered from test executions of other apps (optional input represented with a dashed line in Figure 1) and through input generation. The key idea behind this approach is to generate RTMs that can also be used for executions that are different from the ones considered in this process. We now discuss MOKA's phases in more detail.

2.1 Mock Data Collection

MOKA captures mock data at a predefined set of interaction points, which include locations where the app interacts with its application framework, and focuses on interactions that are common sources of nondeterminism [10]. Specifically, the framework processes interactions that generate network, location, audio, camera, and sensor data. Users can extend this predefined set of interaction points using code annotations. In particular, they can use annotations to instruct MOKA to capture either complete or partial interactions between an app and a third-party library. In the rest of the paper, we generally refer to either framework or third-party library code as *external* code.

At each interaction point, MOKA collects information about the *mocked entity*, the *mock input*, the *mock output*, the *mock components*, and the *mock coverage*. The mocked entity is the part of the software system to be mocked. An example of mocked entity is a method of the application framework called by the app. The data that flows from the app through the interaction point corresponds to the mock input. In the case of an HTTP-based network interaction, for instance, the mock input would consist of the HTTP request (inclusive of its parameters). The mock output captures the data that flows into the app through an interaction point. Using the same HTTP-based network interaction example, the mock output would consist of the HTTP response. The mock components are the objects and methods involved in the execution of external code. Finally, the mock coverage provides information about code coverage in external code. The framework captures mock data through app-level instrumentation and external code tracing.

Algorithm 1: MOKA's test mock generation process.

```
Input  : AUT: app under test
         ts_AUT: test suite for the app under test
         db_app: database of apps
         db_ts: database of test suites for apps in db_app
         T1: developer defined timeout for the mock synthesis process
         T2: developer defined timeout for the input generation process
Output : rtms: reusable test mocks
1  begin
2    rtms = ∅
3    md_ts = Execute-Tests(AUT, ts_AUT)
4    mdMap = Group-By-Mocked-Entity(md_ts)
5    foreach me ∈ mdMap.Key-Set() do
6      md_curr = mdMap.Get(me)
7      rtm_me = null
8      while True do
9        rtm = Perform-Test-Mock-Synthesis(md_curr, T1)
10       if rtm == null then
11         if rtm_me == null then
12           rtm = Create-Record-Replay-Mock(md_curr)
13           rtms.Add(rtm)
14           break
15         else
16           rtms.Add(rtm_me)
17           break
18       else
19         rtm_me = rtm
20         mdi = Collect-New-Mock-Data(me, md_curr, db_app, db_ts, T2)
21         if mdi == null then
22           rtms.Add(rtm_me)
23           break
24         else
25           md_curr.Add(mdi)
26   return rtms
```

2.2 Test Mock Generation

This phase processes mock data to generate reusable test mocks (*RTMs*) by using an iterative process that relies on either program synthesis or record and replay techniques. We describe MOKA's iterative approach with the help of Algorithm 1.

The algorithm takes as inputs (1) the app under test (*AUT*), (2) a test suite for the app under test (ts_{AUT}), (3) an optional database of apps and an optional database of corresponding test suites (db_{app} and db_{ts}), and (4) two timeout values (*T1* and *T2*) that control MOKA's synthesis and input generation steps, respectively. The output of the algorithm is a set of RTMs (*rtms*).

The algorithm starts by executing the test suite associated with the AUT to collect the mock data (md_{ts}) necessary to model interaction points exercised by the tests in the test suite (line 3). After this initial step, MOKA groups mock data by their mocked entity (Group-By-Mocked-Entity); that is, the algorithm groups together mock data flowing to and from the same external code. At this point, MOKA begins its mock generation process by processing the mock data from each mocked entity (lines 5-25).

The iterative mock generation process starts by assigning the value `null` (no mock computed yet) to the mock (rtm_{me}) associated with the mocked entity under analysis (*me*). As we mentioned earlier, MOKA can generate either one of two types of mocks: program-synthesis-based mocks and record-and-replay-based mocks. At a high level, MOKA tries first to generate a mock based on program synthesis; if unsuccessful, it generates a mock based simply on record and replay. The key idea behind favoring program-synthesis-based mocks over record-and-replay-based mocks is that the former can potentially account for mock inputs that were not considered during mock generation, thus creating mocks that can also be reused as the AUT and its test suite evolve.

At each iteration (lines 8-25), MOKA uses the currently available mock data (md_{curr}) to create a test mock (*rtm*) through program synthesis (Perform-Test-Mock-Synthesis). To do so, MOKA leverages and extends a component-based program synthesis algorithm [11]. In a nutshell, component-based program synthesis [12] uses a database of provided components to assemble target programs that are valid in the target language and are consistent with the input-output examples. The specific component-based program synthesis algorithm used by MOKA employs an adaptive search that reuses partial solutions identified in the previous steps of the search process to identify the target program in the space of all possible programs [11]. MOKA extends this algorithm by taking advantage of the fact that the framework already has access to the code that needs to be modeled. Specifically, MOKA reduces the search space by (1) limiting the set of components used in the synthesis process to the set of components associated with the mock data (i.e., mock components from Section 2.1), (2) constraining the search to find a program whose size (in terms of AST nodes) is equal to or less than the size of the method to model, and (3) restricting the composition of components (i.e., composition of AST nodes) to the set of compositions observed in the execution that generated the mock data. The rationale behind these characteristics is to reduce MOKA's search space by disregarding "unlikely" solutions. Finally, MOKA also allows its users to specify a blacklist of components that should not be considered by the algorithm.

If Perform-Test-Mock-Synthesis does not return a valid mock, but MOKA found a valid mock through program synthesis in a previous iteration of the algorithm (line 15), the algorithm saves the mock and proceeds to process the next mocked entity.

Conversely, if Perform-Test-Mock-Synthesis does not return a valid mock (line 10), and a mock was not successfully created in the previous iteration (which can happen only in the first iteration of the algorithm), MOKA creates a simpler mock based on record and replay. This type of mocks only work for inputs that were observed during the mock data collection phase and return the same mock output for a given mock input. They allow MOKA to account for situations in which it is unable to establish a general relationship between mock inputs and outputs, typically because the code representing this relationship is too complex to be synthesized in the provided time budged *T1*. As an example, record-replay-based mocks would suitably model external code that retrieves the first and last name of a student based on the student's university ID.

When Perform-Test-Mock-Synthesis returns a valid test mock (line 18), MOKA stores the mock as its most recent result (line 19), and then proceed to collect new mock data (Collect-New-Mock-Data). MOKA collects new mock data to improve the generality of the mock, that is, its ability to compute the right output when processing a previously unseen (mock) input. The algorithm collects a new mock-data item for *me* per iteration.

MOKA collects new mock data items using two approaches. The first approach runs test cases from a database of apps and corresponding test suites, which are an optional input to the framework. The second approach performs dynamic symbolic execution [13, 14] on the external code based using the currently available mock data. In both cases, MOKA considers a newly generated mock data item as valid only when it increases external code coverage. The framework does so to improve the generality of a mock by exercising new behaviors of external code. If MOKA is not able to find a new mock data item within the time budget *T2*, the framework saves the latest computed mock as the result associated with *me* (line 22). Otherwise, it proceeds to the next iteration of the algorithm and attempts to refine the mock (line 9).

When the algorithm is done processing all the mocked entities, it terminates by returning the computed set of RTMs (*rtms*).

Table 1: Benchmarks used in the preliminary study.

Name	*Category*	*Stars*	*LOC (K)*	*Tests*	*TMs*	*AFTMs*	*ATMs*	*TPLTMs*
CineLog	Multimedia	20	13	152	285	23	210	52
Eventyay	Internet	1342	46	477	268	111	146	11
WiFiAnalyzer	Connectivity	1068	22	708	206	91	85	30
K-9 Mail	Internet	4960	123	536	135	20	104	11
Materialistic	Internet	2010	96	312	97	24	49	24
SMS Backup+	Phone	1499	18	217	75	11	53	11
DNS66	Internet	1447	8	66	60	45	15	0
AnkiDroid	Science	2620	220	248	38	26	11	1
SMSsync	Phone	843	27	23	32	2	16	14
Loop Habit	Sports	3121	39	277	32	0	32	0
Commons	Internet	584	76	21	32	30	2	0
OpenKeychain	Security	1358	130	217	30	15	15	0
Wikipedia	Internet	1171	157	365	24	0	24	0
Web Opac	Reading	118	27	16	23	8	14	1
PageTurner	Reading	451	19	24	20	1	19	0
OpenFoodFacts	Sports	466	251	155	20	2	18	0
FreeOTP	System	653	3	28	19	19	0	0
oandbackup	System	425	14	57	18	3	15	0
Calculate!	Phone	199	18	101	17	9	8	0
AnyMemo	Science	117	34	139	15	1	14	0
				4139	1446	441	850	155

3 PRELIMINARY EMPIRICAL STUDY

To assess the potential usefulness of MOKA, we conducted a preliminary study in which we investigated how developers use test mocks when testing their apps. Specifically, we investigated how many test mocks are used to model the Android framework, the app code, or third-party libraries. To perform the study, we selected the 20 apps with the highest number of test mocks from F-Droid [15] that are also available on GitHub [16]. In order to identify test mocks, we parsed the source code of all apps looking for uses of Mockito [17]—a popularly used framework to manually create test mocks for Android apps [18].

Table 1 lists the apps we used in the study, ordered by their number of tests mocks (column *TMs*). For each app, the table reports the app's name (*Name*), category on F-Droid (*Category*), number of stars on GitHub (*Stars*), size (*LOC (K)*), number of tests (*Tests*), overall number of test mocks (*TMs*), number of test mocks modeling the Android framework (*AFTMs*), number of test mocks modeling code in the app (*ATMs*), and number of test mocks modeling third-party libraries (*TPLTMs*). The table also reports total numbers of *Tests*, *TMs*, *AFTMs*, *ATMs*, and *TPLTMs*.

As the table shows, there are 441 test mocks used to model the Android framework overall, which accounts for 30% of all test mocks. If we also consider test mocks for third-party libraries, the percentage increases to 41%. We believe that these figures provide initial evidence that MOKA has the potential to significantly help developers if it were successful in generating these mocks, or at least part of them, automatically.

In the study, we also assessed how many tests are publicly available for the 1, 220 apps that are on both F-Droid and GitHub and found that about 20% of these apps have tests, for a total of 11, 487

tests. The fact that a large number of tests is available in public repositories, albeit for a relatively small percentage of apps, is consistent with the findings of other related studies [19, 20] and is encouraging; these tests (and their apps) can be used as inputs to MOKA for its test-mock generation process.

3.1 Future Evaluation Plan

We plan to implement MOKA as a tool for the Android platform, by leveraging and extending existing tools that perform app instrumentation [21], record and replay [10], and program slicing [22]. We will then evaluate the framework on real-world apps, starting from the ones considered in our preliminary study. We will use our tool to create RTMs and evaluate whether they can be reused across revisions and with new tests. We will also measure the savings in terms of test running time achieved using MOKA's mocks and assess how many mocks of a given type (i.e., program-synthesis-based vs. record-and-replay-based) MOKA generates.

4 PRACTICAL ADVANTAGES OF MOKA

We believe that MOKA's RTMs will be useful in multiple automated testing scenarios. In particular, MOKA's RTMs could be used to *assist automated input generation*, as they can generate valid mock outputs for previously unseen mock inputs and may allow input-generation techniques to explore the program space efficiently. The ability of handling previously unseen inputs could also help reduce the number of false positives generated by new flaky test executions [2]. For similar reasons, RTMs are expected to be less brittle than traditional mocks, which should allow for (re)using them even as test suites evolve. RTMs could also enable the use of real-world data (which can be collected through record-and-replay-based testing [23]) in cloud-based app testing [24–27]. Additionally, RTMs may be able to make system tests faster and more reliable (e.g., less flaky), thus addressing a significant problem in automated app testing [2, 3].

5 RELATED WORK

Although MOKA is not the first attempt at automating test-mock generation, we believe that it represents a significant leap forward toward having reusable test mocks. In this section, we discuss the work that is most closely related to ours.

AUTOMOCK integrates the creation of mock components with the generation of test cases for various testing goal (e.g., to maximize coverage [6]). Their technique traces post-conditions of mocked methods through symbolic execution to generate new return values for mocked methods. MOKA's approach and goals are different. Our framework uses program synthesis to generate generate mocks that can handle new inputs to mocked entities.

MODA [28] is an extension to PEX that allows for automatically testing applications that use a database by replacing the database with mocks generated through symbolic execution. Given an application and a database schema, MODA produces a parameterized mock that captures the database behavior. Although useful, MODA is tailored to creating database mocks, while our proposed framework can also deal with additional elements of the environment.

Arcuri and colleagues extended Evosuite, a test-input generator, with the ability to generate mock objects for private API calls, so as to improve the coverage of the unit under test [7]. Although using their mocks does improve code coverage, it can also result in a significant number of false alarms. Tillmann and colleagues [29] developed a prototype tool based on symbolic execution that generates mock objects by analyzing all uses of the mock object in a given unit test. Also in this case, because the tool under-approximates the program state, it can result in false positives during testing. MOKA aims to mitigate the problem of false positives by using multiple sources of mock data combined with program synthesis. In addition, MOKA also proposes a solution for generating mocks that can suitably handle previously unseen inputs.

Saff and Ernst [8, 30], Joshi and Orso [31], and Elbaum and colleagues [32] presented variations on the idea of factoring tests to speed up system test executions [30]. They discuss the idea of tracing values across system boundaries to extract tests that can be run in isolation together with their necessary scaffolding. Although MOKA is related to these approaches, it goes one step forward by generating, when possible, general mocks that can also be used in the presence of previously unseen mock inputs.

Qi and colleagues [9] developed a technique to construct models for library and system call functions using program synthesis and a set of predefined components. MOKA leverages program synthesis as well, but it automatically identifies the set of components for the synthesis task, making the approach more generally applicable.

Samimi and colleagues proposed the idea of declarative mocking [33], in which developers write method specifications for the API being mocked in a high-level logical language, and a constraint solver dynamically executes these specifications upon method invocation. Similarly, Galler and colleagues presented an approach for automatically deriving the behavior of mock objects from given design-by-contract specifications [34]. Finally, Android Studio (https://developer.android.com/studio/) offers some support for manually configuring mocks for mobile apps [35]. MOKA aims to overcome the main limitation of these techniques, that is, that developers have to write method specifications or manually design mock configurations in order to generate and use test mocks.

6 CONCLUSIONS

We proposed MOKA, a framework for generating reusable test mocks for mobile apps by leveraging existing test executions. Our framework analyzes the input-output relationships at the interface between the app under test and its environment and tries to generate test mocks that are highly reusable, yet accurate. We also presented a preliminary study showing MOKA's potential impact, described our plan to evaluate our envisioned framework, and discussed how multiple automated testing scenarios could benefit from the test mocks generated by MOKA. Our immediate next steps towards the realization of our vision involve exploring trade-offs between reusability and accuracy of the generated mocks and investigating the use of MOKA in the context of program evolution.

ACKNOWLEDGMENTS

This work was partially supported by a gift from Facebook, NSF grant CCF-1563991, Spanish Government's SCUM grant RTI2018-102043-B-I00, and the Madrid Regional project BLOQUES.

REFERENCES

[1] S. R. Choudhary, A. Gorla, and A. Orso, "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 (ASE)*. IEEE Computer Society, 2015, pp. 429–440.

[2] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, 2018, pp. 1–23.

[3] S. Thorve, C. Sreshtha, and N. Meng, "An empirical study of flaky tests in android apps," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE Computer Society, 2018, pp. 534–538.

[4] M. Fowler. (2006, Jan.) Testdouble. [Online]. Available: https://www.martinfowler.com/bliki/TestDouble.htmll

[5] G. Meszaros, *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[6] N. Alshahwan, Y. Jia, K. Lakhotia, G. Fraser, D. Shuler, and P. Tonella, "AUTOMOCK: automated synthesis of a mock environment for test case generation," in *Practical Software Testing: Tool Automation and Human Factors, 14.03. - 19.03.2010*, 2010.

[7] A. Arcuri, G. Fraser, and R. Just, "Private api access and functional mocking in automated unit test generation," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2017, pp. 126–137.

[8] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, "Automatic test factoring for java," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. Association for Computing Machinery, 2005, pp. 114–123.

[9] D. Qi, W. N. Sumner, F. Qin, M. Zheng, X. Zhang, and A. Roychoudhury, "Modeling software execution environment," in *2012 19th Working Conference on Reverse Engineering*. Kingston, Ontario, Canada: IEEE Computer Society, 2012, pp. 415–424.

[10] Y. Hu, T. Azim, and I. Neamtiu, "Versatile yet lightweight record-and-replay for android," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2015, pp. 349–366.

[11] K. Shi, J. Steinhardt, and P. Liang, "Frangel: component-based synthesis with control structures," *Proceedings of the ACM on Programming Languages*, pp. 1–29, 2019.

[12] S. Gulwani, O. Polozov, R. Singh *et al.*, "Program synthesis," *Foundations and Trends® in Programming Languages*, pp. 1–119, 2017.

[13] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 213–223.

[14] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," in *Proceedings of the 10th European Software Engineering Conference / 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005, pp. 263–272.

[15] (2020) F-droid. [Online]. Available: https://f-droid.org

[16] (2020, Jan.) Github. [Online]. Available: https://github.com

[17] (2020) Tasty mocking framework for unit tests in java. [Online]. Available: https://site.mockito.org

[18] (2020) Build local unit tests. [Online]. Available: https://developer.android.com/training/testing/unit-testing/local-unit-tests

[19] F. Pecorelli, G. Catolino, F. Ferrucci, A. D. Lucia, and F. Palomba, "Testing of Mobile Applications in the Wild: A Large-Scale Empirical Study on Android Appsy," in *Proceedings of the 28th IEEE/ACM International Conference on Program Comprehension*, 2020.

[20] J.-W. Lin, N. Salehnamadi, and S. Malek, "Test automation in open-source android apps: A large-scale empirical study," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020.

[21] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, 1999.

[22] T. Azim, A. Alavi, I. Neamtiu, and R. Gupta, "Dynamic slicing for android," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1154–1164.

[23] M. Fazzini, E. N. D. A. Freitas, S. R. Choudhary, and A. Orso, "Barista: A technique for recording, encoding, and running platform independent android tests," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Tokyo, Japan: IEEE Computer Society, 2017, pp. 149–160.

[24] J. Dunn, A. Mols, L. Lomax, and P. Medeiros. (2017, May) Managing resources for large-scale testing. [Online]. Available: https://code.facebook.com/posts/1708075792818517/managing-resources-for-large-scale-testing

[25] Amazon. (2019, May) Aws device farm. [Online]. Available: https://aws.amazon.com/device-farm

[26] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. Association for Computing Machinery, 2016, p. 94–105.

[27] N. Alshahwan, X. Gao, M. Harman, Y. Jia, K. Mao, A. Mols, T. Tei, and I. Zorin, "Deploying search based software engineering with sapienz at facebook," in *Search-Based Software Engineering*. Cham: Springer International Publishing, 2018, pp. 3–45.

[28] K. Taneja, Y. Zhang, and T. Xie, "Moda: Automated test generation for database applications via mock objects," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, 2010, p. 289–292.

[29] N. Tillmann and W. Schulte, "Mock-object generation with behavior," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. IEEE Computer Society, 2006, pp. 365–368.

[30] D. Saff and M. D. Ernst, "Mock object creation for test factoring," in *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. Association for Computing Machinery, 2004, pp. 49–51.

[31] S. Joshi and A. Orso, "Capture and Replay of User Executions to Improve Software Quality," Georgia Institute of Technology – College of Computing, Tech. Rep. 1006-04-11, April 2006, http://www.cc.gatech.edu/~orso/papers/abstracts.html#joshi06apr-tr.

[32] S. Elbaum, H. N. Chin, M. Dwyer, and J. Dokulil, "Carving differential unit test cases from system test cases," in *Proc. FSE*, 2006, pp. 253–264.

[33] H. Samimi, R. Hicks, A. Fogel, and T. Millstein, "Declarative mocking," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. Association for Computing Machinery, 2013, pp. 246–256.

[34] S. J. Galler, A. Maller, and F. Wotawa, "Automatically extracting mock object behavior from design by contract™ specification for test data generation," in *Proceedings of the 5th Workshop on Automation of Software Test*. Association for Computing Machinery, 2010, pp. 43–50.

[35] (2020) Configure on-device developer options. [Online]. Available: https://developer.android.com/studio/debug/dev-options