RIFDiscoverer: A Tool for Finding Resource Interaction Failures

Euler Marinho
Federal University of Minas Gerais
Belo Horizonte, Minas Gerais, Brazil

Izaias Machado Pessoa Neto
Federal University of Ceará
Sobral, Ceará, Brazil

Fischer Ferreira
Federal University of Itajubá
Itabira, Minas Gerais, Brazil

João P. Diniz
Federal University of Minas Gerais
Belo Horizonte, Minas Gerais, Brazil

Eduardo Figueiredo
Federal University of Minas Gerais
Belo Horizonte, Minas Gerais, Brazil

ABSTRACT

Mobile application quality has become a crucial aspect. In spite of the numerous studies concerning testing methods, tools, and techniques, the field of mobile application testing is still under development. In mobile applications, resource interaction failures occur when resources influence the behavior of other resources. They can compromise the application and harm the user experience. This paper presents RIFDiscoverer a tool for assisting developers and testers to deal with resource interaction failures in Android applications. Our preliminary results indicates a potential of the tool to find resource interaction failures. The source code and a demo video are available on GitHub at https://github.com/byte-skiing/RIFDiscoverer.

CCS CONCEPTS

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

KEYWORDS

Software Quality, Software Testing, Resource Interaction Failures

ACM Reference Format:
Euler Marinho, Izaias Machado Pessoa Neto, Fischer Ferreira, João P. Diniz, and Eduardo Figueiredo. 2024. RIFDiscoverer: A Tool for Finding Resource Interaction Failures. In . ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Mobile applications have been developed not only for entertainment purposes but also for targeting safety and critical domains [3]. As a consequence, the quality of mobile applications has become a crucial aspect, for instance, by promoting the use of testing as a quality assurance practice [15, 16]. However, despite the availability of testing methods, techniques, and tools, the field of mobile application testing is still under development [9]. These applications are often executed on a variety of platform configurations [11] and each platform configuration has different platform resources. These resources may be related to communication capabilities (e.g., Wi-Fi, Bluetooth, Mobile Data, and GPS), sensors (e.g., Accelerometer, Gyroscope, and Magnetometer), and user-controlled options (e.g., Battery Saver and Do Not Disturb). However, applications can present unexpected behaviors since the resource interactions can introduce failures that manifest themselves in specific resource combinations [20].

Resource interaction occurs when one or more resources influence the behavior of other resources, similarly to the feature interaction problem in configurable software systems [4, 5] and telecommunication systems [7]. An example of resource interaction failure occurs for Wikimedia Commons app when a pair of resources are disabled [24]. The high number of input combinations is a challenging aspect for testing software systems in general, since the effort of the exhaustive testing is generally prohibitive. For instance, it is the case of configurable systems [4, 8, 10] in which all tests must be executed in several configurations. An alternative for decreasing the testing effort is the use of sampling strategies involving the selection of a subset of input combinations. Sampling strategies are a well known technique, such as in the domain of configurable systems [4]. They have been shown to be effective in finding faults, even with the number of combinations tested much lower than the universe of all possible combinations [10, 21, 23].

In this paper, we introduce RIFDiscoverer, a tool that helps developers and testers to deal with resource interaction failures in Android applications. We have chosen a simple but responsive user interface for RIFDiscoverer. By using an extensible architecture, it can be used in research efforts aiming to deal with scalability factors, for example, when interacting with physical or virtual device farms [18]. Our preliminary results indicates a potential of the tool to find resource interaction failures.

This paper is organized as follows. In Section 2, we discuss background information and some related work. In Section 3, we present the architecture, some design and implementation aspects of RIFDiscoverer. Section 4 presents a preliminary evaluation of the tool. Section 5 presents some concluding remarks.

2 BACKGROUND AND RELATED WORK

This section discusses resource interaction failures (Section 2.1) and presents an overview of sampling testing strategies (Section 2.2).

2.1 Resource Interaction Failures

We define "resource interaction failures" as failures that occur when resources influence the behavior of other resources. This definition is inspired by the feature interaction problem in configurable systems [4]. Our study includes 12 resources often used by Android applications and present in most devices: `Location`, `Wi-Fi`,

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

`Mobile Data`, `Bluetooth`, `Auto Rotate`, `Battery Saver`, `Do Not Disturb`, `Accelerometer`, `Gyroscope`, `Magnetometer`, `Proximity`, and `Camera`.

Figure 1 presents a code excerpt of `Wikimedia Commons Android` app, showing a case of a resource interaction failure [24]. This open source application allows users to upload pictures from the device to Wikimedia Commons, the image repository for Wikipedia[1]. The Android Platform supports the positioning via GPS or network (Wi-Fi/Mobile Data). The issue describes a situation involving the crash of the application when it is opened and both GPS and network are disabled [6]. The failure is caused by the call of `getLastKnownLocation` to get the current location via network (line 3). However, this call returns a `null` value which is later used in the construction of an object to store the location-related values (line 5). As a result, the application crashes because of a `NullPointerException`.

```
1  locationManager.getLastKnownLocation(
       LocationManager.GPS_PROVIDER);
2  if (lastKL == null) {
3   lastKL = locationManager.getLastKnownLocation
       (LocationManager.NETWORK_PROVIDER);
4  }
5  return LatLng.from(lastKL); //An object is
       constructed from the latitude and
       longitude coordinates
```

Figure 1: Code Excerpt from Wikimedia Commons app.

2.2 Sampling Testing Strategies

As the exhaustive exploration of the input space in configurable software systems is often very expensive or even impractical (for instance, by brute-force), an alternative to balance the effort and the failure-detection capability is to use sampling testing strategies [1, 10, 21, 23]. The use of sampling testing has been promising to find feature interaction failures [1, 10, 21, 23] and resource interaction failures [20]. For instance, the strategy One-Disabled [1] selects settings with only one resource disabled and all other resources enabled. The strategy One-Enabled selects settings with only one resource enabled and the other resources disabled. The strategy Most-Enabled-Disabled combines two sets of samples: one set in which most of the resources are enabled and other set in which most of the resources are disabled. In the case when constraints between resources do not exist, it establishes two settings: one with all resources enabled and one with all resources disabled [21]. The strategy Random creates n distinct settings with all resources randomly enabled or disabled. We used the implementation of this strategy present in FeatureIDE [25].

2.3 Related Work

Table 1 provides a brief comparison between our tool and tools of similar previous studies. In each case, the columns refer to the individual studies. The rows show the characteristics of the studies, such as which strategy or technique was used. We use a dash '-' to indicate information not available in the respective study. To our knowledge, RIFDiscoverer is a tool that explicitly works with a wide range of device sensors.

3 THE RIFDISCOVERER TOOL

This section provides a comprehensive overview of the RIFDiscoverer architecture (Section 3.1), design and implementations concerns taken into consideration (Section 3.2) and the instrumentation used for testing (Section 3.3).

3.1 Architecture

Figure 2 presents the general architecture of RIFDiscoverer. The application requires the user to choose one available testing strategy (One-Disabled, One-Enabled, Most-Enabled-Disabled, Pairwise, Random, Custom, IncLing [2], CASA [12], Chvatal [14], ICPL [13]) and specify the path where the instrumented Android project is located. Once the testing strategy and execution parameters are defined, the application communicates with the WebSocket Server to start the execution of the tests. Before test execution, the WebSocket Server starts Genymotion Android emulator. Test Execution Manager receives all settings and the number of executions, and then it executes them one by one. Note that for each execution, the order in which the settings are processed is random. The setting processing script is responsible for executing the Gradle task that runs the test suite, with only the enabled resources for the current setting on the emulated device. After finished running the tests for a setting, the WebSocket Server sends a message to the Application to update a progress bar and the same is done for the following setting.

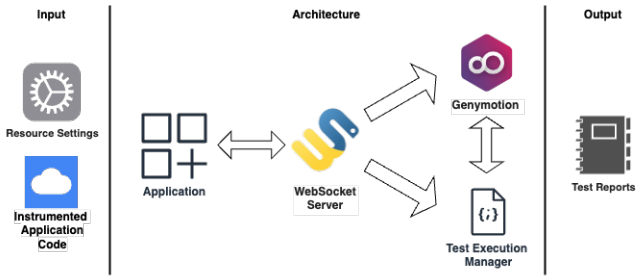

Figure 2: Architecture of RIFDiscoverer.

3.2 Design and Implementation

The two key components of RIFDiscoverer are the application front-end through which the user interacts with the tool and a WebSocket Server that rapidly runs scripts and communicates with the other application components.

For the front-end, we aim for an easy to use interface and a way for the interface to communicate directly to Python running on the user machine. For this reason, we choose Python EEL[2]. This framework allows for the development of a web-based front-end while maintaining a Python back-end. It facilitates seamless

[1]https://commons-app.github.io/

[2]https://github.com/python-eel/Eel

Table 1: Comparison of Related Tools

	RIFDiscoverer	FicFinder [27]	PREFEST [19]	Vilkomir's Tool [26]	SetDroid [24]
Operating System	Android	Android	Android	Android	Android
Strategies-Techniques	Sampling	Static Analysis	Pairwise	Each-Choice	Setting-wise metamorphic fuzzing
Comparison Elements (Resources, hardware options)	14	-	4	5	> 50

interaction between the user interface and the underlying Python code, enabling the front-end to directly call Python implemented functions.

WebSocket Server is a separate component that uses Python WebSockets[3] library and runs alongside EEL, but in a separate thread. This design was implemented to handle the execution of scripts that takes a long time to complete, ensuring the application remains responsive throughout the execution of testing strategies. By running in a separate thread, WebSocket Server can manage lengthy operations without blocking the application thread. This approach ensures that users receive updates and can interact with the interface even during extensive testing processes.

3.3 Test Instrumentation

We implemented a test instrumentation based on the UI Automator framework[4] to control the resources. The instrumentation is based on Android instrumented tests, i.e., a type of functional test[5]. They execute on devices or emulators and can interact with Android framework APIs. We are to manage 7 resources interacting with Android Quick Settings: Auto Rotate, Battery Saver, Bluetooth, Do Not Disturb, Location, Mobile Data, and Wi-Fi. We control the other resources using third-party applications. For instance, Camera is controlled by Lens Cap[6] and the sensors (Accelerometer, Gyroscope, Magnetometer, Proximity) are managed by Sensor Disabler[7]. The test instrumentation consists of the function AdjustResourceStates presented in Algorithm 1. For the required setting S, we enable (line 6) or disable (line 8) each resource state (line 4) according to the state specified in the pair.

We implemented Resource_setup as a class with a static method annotated with BeforeClass[8]. We extended each class of the test suites with the implemented class. Therefore, the execution of tests of a certain class is preceded by the execution of the setup method. In the current implementation, we perform the verification of resource state (line 5) via Android APIs, such as LocationManager[9] for the `Location` and TelephonyManager[10] for the `Mobile Data`. In other cases, we use the UI Automator features to find some screen widgets related to the resource state. For example, we inspect the sensors states by processing screens of Sensor Disabler. It is important to emphasize that in our implementation the resources are only adjusted (lines 6 and 8) if necessary. Besides, we modified the build scripts in order to use the Android Test Orchestrator[11], a tool that helps minimize possible shared states, a known factor associated to flaky tests [22] and isolate the crashes.

Algorithm 1 Resource_setup

```
1:  Input
2:      S     list of < resource, state > pairs
3:  procedure AdjustResourceStates(S)
4:      for all pair ∈ S do
5:          if pair.state == true then
6:              enable(pair.resource)
7:          else
8:              disable(pair.resource)
9:          end if
10:     end for
11: end procedure
```

We implemented Algorithm 2 for managing the executions of the instrumented test suites. We used three executions (line 6) to deal with flaky tests, and shuffled the settings to minimize order dependencies between tests (line 8). Multiple execution is a common strategy for detecting flaky tests. However, the optimal number of re-executions to identify flaky tests is not defined [22]. One study suggests a maximum of five re-executions [17]. Based on previous essays, we set the default number of re-executions to three. We took into account our time constraints for the experiments and made the observation that this number is sufficient to detect flaky tests. We call the function AdjustResourceStates (line 10) defined in Algorithm 1 to adjust the states of all resources.

4 PRELIMINARY EVALUATION

To evaluate the usefulness of RIFDiscoverer, we applied it to Threema[12], a messenger focused on privacy and security. Table 2 presents the characteristics of the study. The declared resources are identified from the static analysis of the Android Manifest file. The tool takes as input settings, i.e. lists of pairs `(resource,state)` where `state` can be `True` or `False` depending on whether the `resource` is enabled or disabled. Our preliminary found a one resource interaction failure in Threema. The failure we found (by a test named *"testNotificationWithoutAction"*) happens when Do Not Disturb is enabled. This failure could let the user to loose an error notification, harming its experience.

[3]https://github.com/python-websockets/websockets
[4]https://developer.android.com/training/testing/ui-automator
[5]https://developer.android.com/training/testing/instrumented-tests
[6]https://github.com/percula/LensCap
[7]https://github.com/wardellbagby/Sensor-Disabler
[8]https://junit.org/junit4/javadoc/4.12/org/junit/BeforeClass.html
[9]https://developer.android.com/reference/android/location/LocationManager
[10]https://developer.android.com/reference/android/telephony/TelephonyManager
[11]https://developer.android.com/training/testing/junit-runner
[12]https://github.com/threema-ch/threema-android

Algorithm 2 Test_execution_manager

```
Input
    AP  application with extended tests
    SL  list of settings
Output
    TR  test reports
maxExecutions ← 3 //default
for exec ← 1 to maxExecutions do
    SHUFFLE(SL)
    for all st ∈ SL do
        ADJUSTRESOURCESTATES(st)
        Execute the whole test suite of AP
    end for
end for
```

Table 2: Evaluation of Threema Case Study

Characteristic	Description
LOC	238,045
Test LOC	1,931
Test Cases	54
Settings	Random(30), One-Disabled(12), One-Enabled(12), Pairwise(8), Most-Enabled-Disabled(2)
Declared Resources	Bluetooth, Camera, Location, Mobile data, Wi-Fi
Failed Test Cases	1

5 CONCLUSION

This paper presented RIFDiscoverer, a tool that helps developers and testers to deal with resource interaction failures. We have chosen a simple but responsive user interface for RIFDiscoverer. We described its architecture, design and implementation aspects, and performed a preliminary evaluation. By using an extensible architecture, it can be used in research efforts aiming to deal with scalability factors, for example, when interacting with physical or virtual device farms [18]. We plan further improvements on RIFDiscoverer implementation. Currently, we are planning an adaptation of the tool as an Web Application to be hosted in cloud computing platforms. Future studies can involve empirical studies with application developers, deal with other kinds of resources, and the portability of the tool to other mobile platforms.

REFERENCES

[1] I. Abal, C. Brabrand, and A. Wasowski. 2014. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 421–432.

[2] M. Al-Hajjaji, S. Krieter, T. Thüm, M. Lochau, and G. Saake. 2016. IncLing: efficient product-line testing using incremental pairwise sampling. *ACM SIGPLAN Notices* 52, 3 (2016), 144–155.

[3] D. Amalfitano, N. Amatucci, A. M. Memon, P. Tramontana, and A. R. Fasolino. 2017. A general framework for comparing automatic testing techniques of Android mobile apps. *Journal of Systems and Software (JSS)* 125 (2017), 322–343.

[4] S. Apel, D. Batory, C. Kastner, and G. Saake. 2013. *Feature-oriented software product Lines*. Springer Berlin / Heidelberg.

[5] S. Apel, S. Kolesnikov, N. Siegmund, C. Kästner, and B. Garvin. 2013. Exploring Feature Interactions in the Wild: The New Feature-Interaction Challenge. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development (FOSD)*. 1––8.

[6] Commons App. [n. d.]. Commons Issue 1735. https://github.com/commons-app/apps-android-commons/issues/1735, Accessed 6-mar-2024.

[7] T. F. Bowen, F. S. Dworack, C. Chow, N. Griffeth, G. E. Herman, and Y-J Lin. 1989. The feature interaction problem in telecommunications systems. In *Proceedings of the 7th International Conference on Software Engineering for Telecommunication Switching Systems (SETSS)*. 59–62.

[8] M. B. Cohen, M. B. Dwyer, and J. Shi. 2007. Interaction Testing of Highly-Configurable Systems in the Presence of Constraints. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 129–139.

[9] C. Escobar-Velásquez, M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk. 2020. Enabling Mutant Generation for Open- and Closed-Source Android Apps. *IEEE Transactions on Software Engineering (TSE)* 48, 1 (2020), 186–208.

[10] F. Ferreira, G. Vale, J. P. Diniz, and E. Figueiredo. 2021. Evaluating T-wise testing strategies in a community-wide dataset of configurable software systems. *Journal of Systems and Software (JSS)* (2021), 110990.

[11] J. A. Galindo, H. Turner, D. Benavides, and J. White. 2016. Testing variability-intensive systems using automated analysis: an application to Android. *Software Quality Journal* 24 (2016), 365–405.

[12] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. 2011. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering* 16, 1 (2011), 61–102.

[13] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2011. Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In *International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 638–652.

[14] Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Anne Grete Eldegard, and Torbjørn Syversen. 2012. Generating Better Partial Covering Arrays by Modeling Weights on Sub-product Lines. In *International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 269–284.

[15] M. C. Júnior, D. Amalfitano, L. Garcés, A. R. Fasolino, S. A Andrade, and M. Delamaro. 2022. Dynamic Testing Techniques of Non-functional Requirements in Mobile Apps: A Systematic Mapping Study. *ACM Computing Surveys (CSUR)* 54, 10s (2022), 1–38.

[16] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein. 2018. Automated testing of Android apps: A systematic literature review. *IEEE Transactions on Reliability* 68, 1 (2018), 45–66.

[17] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov. 2020. Understanding reproducibility and characteristics of flaky tests through test reruns in Java projects. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*. 403–413.

[18] Hao Lin, Jiaxing Qiu, Hongyi Wang, Zhenhua Li, Liangyi Gong, Di Gao, Yunhao Liu, Feng Qian, Zhao Zhang, Ping Yang, and Tianyin Xu. 2023. Virtual Device Farms for Mobile App Testing at Scale: A Pursuit for Fidelity, Efficiency, and Accessibility. In *Proceedings of the Annual International Conference on Mobile Computing and Networking*. Article 45, 17 pages.

[19] Y. Lu, M. Pan, J. Zhai, T. Zhang, and X. Li. 2019. Preference-wise testing for Android applications. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 268–278.

[20] E. H. Marinho, F. Ferreira, J. P. Diniz, and E. Figueiredo. 2023. Evaluating testing strategies for resource related failures in mobile applications. *Software Quality Journal* (2023), 1–27.

[21] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel. 2016. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 643–654.

[22] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. 2021. A Survey of Flaky Tests. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1, Article 17 (2021), 74 pages.

[23] S. Souto, M. d'Amorim, and R. Gheyi. 2017. Balancing soundness and efficiency for practical testing of configurable systems. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 632–642.

[24] J. Sun, T. Su, J. Li, Z. Dong, G. Pu, T. Xie, and Z. Su. 2021. Understanding and Finding System Setting-Related Defects in Android Apps. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 204–215.

[25] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79 (2014), 70–85.

[26] Sergiy Vilkomir. 2018. Multi-device coverage testing of mobile applications. *Software Quality Journal (SQJ)* 26, 2 (2018), 197–215.

[27] L. Wei, Y. Liu, S-C. Cheung, H. Huang, X. Lu, and X. Liu. 2020. Understanding and Detecting Fragmentation-Induced Compatibility Issues for Android Apps. *IEEE Transactions on Software Engineering (TSE)* 46, 11 (2020), 1176–1199.

A SCREENS

Figure 3 presents the first screen of RIFDiscoverer. This screen shows fields to the selection of testing strategies (A) and a "T"

parameter (B) depending on the selected strategy. A part of the screen (C) presents a table of settings with markers for enabled and disabled resources. Some strategies (Random and Custom) allow direct manipulation of the settings.

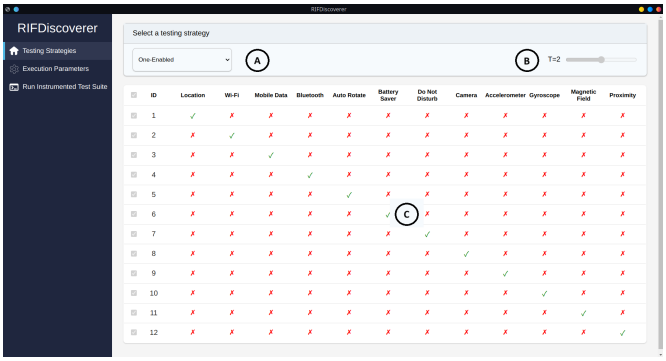

Figure 3: Choice of Testing Strategies.

Figure 4 presents the second screen of RIFDiscoverer. This screen allows changing some execution parameters: the folders of application project and generated reports of Gradle verification task, the number of executions for dealing with flaky tests, and the Gradle task name related to the execution of the instrumented test suite.

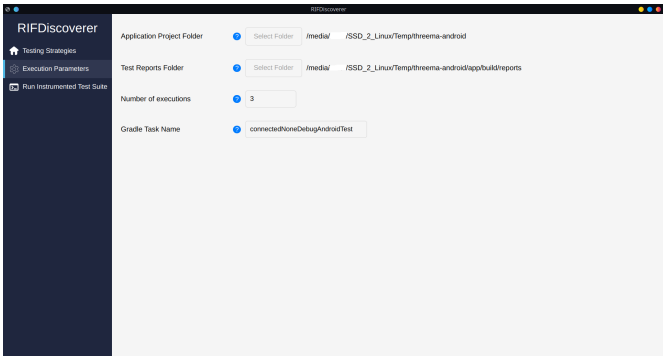

Figure 4: Changing Execution Parameters.

Figure 5 presents the third screen of RIFDiscoverer. This screen allows managing the execution of the instrumented test suite. A part of the screen (A) is a terminal with output logs from test suite executions. A button (B) loads the Genymotion emulator preconfigured to the execution of instrumented code. Another button (C) starts the execution of the test suite.

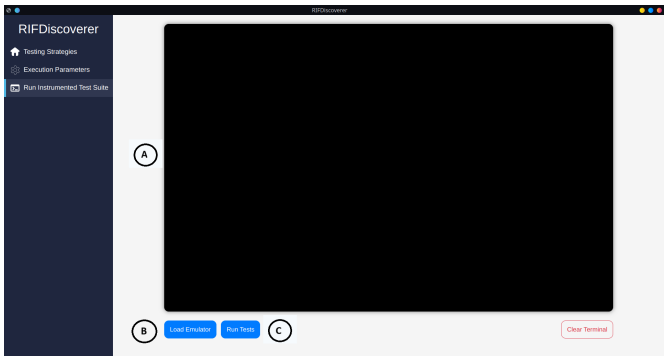

Figure 5: Running the Test Suite.

Figure 6 presents the third screen of RIFDiscoverer side by side with the emulator. The screen presents a progress bar (A) and buttons to abort the execution (B) and visualize the folder of the test reports (C).

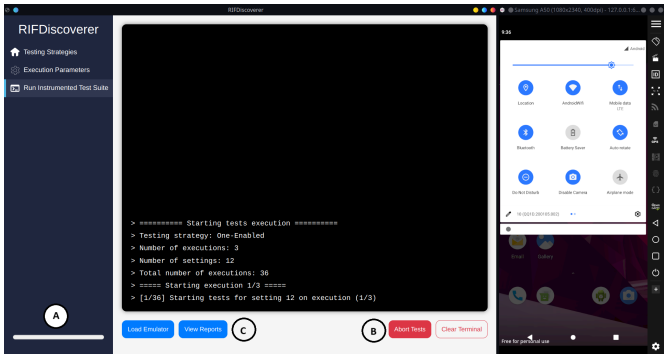

Figure 6: Running the Test Suite with Emulator Loaded.