Detecting anomalous device loads during exploratory testing of mobile applications

Danila Mikhaltsov
Ivannikov Institute for System Programming of the RAS
Moscow, Russia
dmikhaltsov@ispras.ru
ORCID: 0000-0001-6696-0643

Konstantin Sorokin
Ivannikov Institute for System Programming of the RAS
Moscow, Russia
ksorokin@ispras.ru
ORCID: 0000-0002-6861-3802

***Abstract*—Reputation and competitiveness of both mobile applications and mobile operating systems depend on their quality. Developers are using various techniques to ensure high quality. Recently, exploratory testing approaches have been gaining significant attention in this context. However, these approaches mostly do not consider one major non-functional requirement that affects quality – performance issues. Performance issues, such as sluggish UI or extensive battery consumption, are tightly connected to inefficient use of device resources. Detecting these issues is a non-trivial task. In this paper we present a novel approach to detect anomalous device resource usage and hence potential performance issues. Our approach is integrated with exploratory testing framework and uses information about previously executed test runs to build the expected resource usage model. The underlying model represents resource usage data as a multidimensional time series and is able to detect anomalous time intervals. We integrate our approach with exploratory testing tool for Android and empirically evaluate it on a set of real-world applications with injected performance issues. Our results show that suggested approach can be successfully applied to detect anomalous device resource usage and potential performance regressions.**

***Index Terms*—Performance loss, Mobile applications, Anomaly detection, Regression analysis, Performance regression, Android**

I. Introduction

In the last decade mobile application market has seen fast growth and now it is a key business segment for many companies. For example, Android developed by Google is one of the most popular mobile operating systems. It holds 75% of the mobile application market and has more than 2.8 billion active users. The users' interaction with the mobile device is performed via applications. Both modern mobile applications and mobile operating systems are complex software products that require significant resources for development, support and quality assurance. Despite these efforts mobile software still contains bugs and errors.

Various practices to find different types of errors, including application testing methods, are used. Application testing is often divided into functional and non-functional. The first type of testing is used to verify correctness of the application and compliance with functional requirements. The second one is used for checking non-functional requirements. For example, one can search for inefficient network utilization or UI layout issues. These types of problems arise when an application has *performance bugs*.

The value of finding such errors lies in the fact that they greatly affect the end-user experience. Applications with greater stability are more competitive, so correcting such errors is extremely important for application developers. According to studies [1], [2], performance bugs survive long in the system and are more difficult to detect and fix compared to other bugs.

Recently, approaches to automatically test mobile applications via interacting with application UI are gaining significant attention [3]–[6]. This type of testing is called GUI testing or *exploratory testing*. The goals of such testing may vary but most often GUI testing aims to find crashes or hangs by exploring as many application states as possible in the limited amount of time.

During exploratory testing of a mobile application, *resource usage data* can be collected: CPU usage (per process and overall), memory usage, power consumption, the number of system calls, etc. Recently, anomaly detection on this data for non-functional testing of applications has gained popularity [7]–[9]. Resource usage data can be utilized to find *anomalous device loads*, i.e. increased CPU usage or network usage, etc. The anomalous device loads, in turn, may indicate presence of performance bugs.

There are various tools for exploratory and dynamic testing of Android applications, e.g. Android Studio profilers, Monkey [10], DroidBot [3], etc. However, to the best of our knowledge, there are no existing tools that try to detect anomalous resource utilization (or performance issues) during exploratory testing of mobile applications. This paper aims to mitigate this shortcoming.

We propose an approach to detect anomalous device resource utilization by tracking history of exploratory testing runs and using them to build the expected resource utilization model. Thus, we consider this problem a part of *regression* exploratory testing.

The paper is structured as follows. First, Section II provides the necessary background with overview of related concepts. Then, in Section III we describe our approach in details. It contains an overview of testing pipeline, information about resource usage data collecting and processing and in-depth explanation of our anomaly detection approach. It is followed by evaluation with experiments on real-world applications (Section IV). Finally, sections V and VI contain a short summary of related work and conclusion respectively.

II. Background

This section contains a short summary of related concepts and practices.

A. GUI testing

Automated *GUI testing* or *exploratory testing* approaches try to simulate real user by interacting with mobile application UI. The goals of such testing may vary but most often it aims to find application crashes or hangs by exploring as many application states as possible in the limited amount of time. Numerous GUI testing approaches have been proposed by scientific community [3]–[6]. They utilize different methods, but the high-level workflow is mostly the same. Typically there is a testing agent which interacts with a real mobile device or with a mobile emulator. This agent sends actions to the device, e.g. touch or scroll, and receives application state updates if necessary, which may affect next action choice. This is illustrated in the upper part of Figure 1.

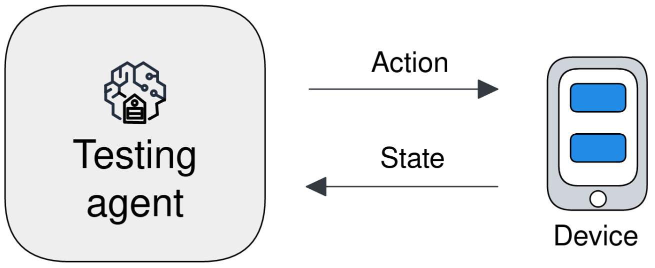

Fig. 1: GUI testing high-level workflow

To start testing user provides an application and *testing (exploration) configuration*, which includes things such as test session time, exploration strategy (e.g. how actions are selected), etc. The output of completed GUI testing session defines a finished *test (exploration) run*.

B. Performance regressions

During software evolution some changes may result in performance regressions. Performance regression can be defined as a degradation of the analyzed system compared to its previous versions. To detect such regressions, developers conduct performance regression testing, which is based on collection of resource usage data and performance metrics during each test run. Resource usage data is generally represented as multidimensional time series, e.g. joint time series of CPU usage, memory usage or storage utilization by target application under test. This data is typically stored in a dedicated database, which can be later queried for detailed analysis.

Once new software version is available, one can execute the available performance tests and compare newly collected data with the history of previous test executions. If data for new version deviates too much from the expected performance or from previously observed behavior, a performance regression is reported.

C. Time series anomalies

Time series anomaly detection problem is usually formulated as identifying outlier data points relative to some expected behavior. Because time series is a sequence of points correlated in time, outliers can be divided into two types: point outliers and sub-sequence outliers.

A point outlier can be visualized as an unexpected spike or drop in time series. It is a point value in a sequence that looks unusual at a specific time when compared either to the other values in the time series (global outlier), or to its neighboring points (local outlier). One can distinguish between univariate and multivariate point outliers.

Sub-sequence outliers represent consecutive points in time whose joint behavior is unusual. However, it does not mean that each observation individually has necessarily to be a point outlier. Sub-sequence outliers can also be global or local, and univariate or multivariate.

Finally, the whole time series can be considered anomalous (or an outlier) when compared to a set of other time series. This can happen in cases when the target time series has:

- different length or time shift
- different distribution of values (but not necessarily different time-series curve shape)
- different time-series curve shape (but not necessarily different distribution of values)

In the context of performance testing we consider all these types of anomalies important. For example, one can observe occasional point or sub-sequence anomalies in CPU-utilization, differences in curve shape of network traffic usage or differences in distribution of memory residual set size (RSS).

III. Approach

In this section, we describe our approach for detecting abnormal loads on a device during GUI testing in detail. We first present the overview of our approach and explain overall pipeline. After that, we explain how resource usage data is collected and post-processed, how we build an expected model and how it is used to detect anomalies.

A. Overview

We combine regression testing (in particular performance regression testing) with automated GUI testing for assessing mobile application or mobile OS quality. Figure 2 contains schematic diagram of the overall pipeline.

The regression testing system supports two complementary scenarios. In the first scenario, testing is performed on subsequent versions $(v_1, v_2 \ldots, v_n)$ of the same mobile application. In the second scenario incremental mobile OS versions can be evaluated on a set of predefined applications. There is no difference in the overall pipeline for both these cases.

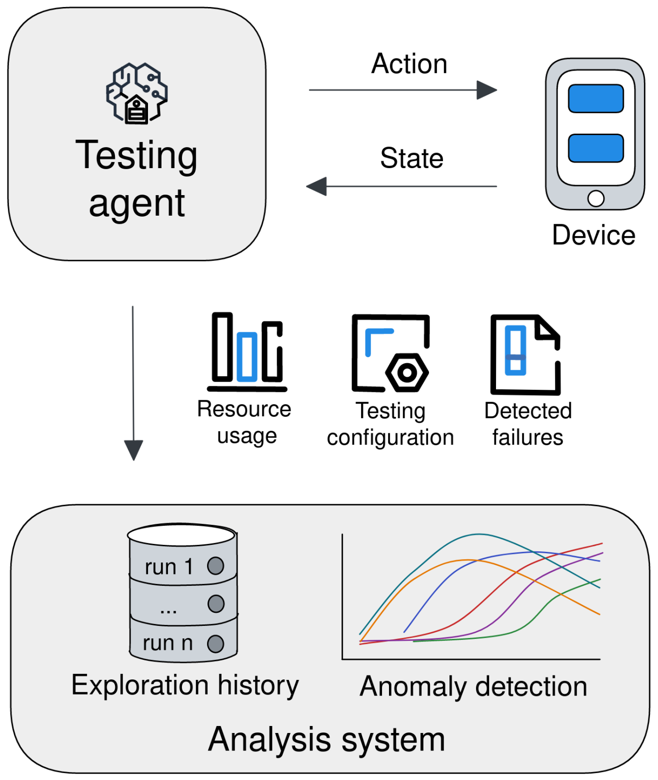

Fig. 2: Overall pipeline

Each time new version is uploaded to the regression testing system, the application is run using an automated exploratory testing tool. This is usually performed with a set of fixed configurations $\mathcal{C} = (D, S, T, I)$, where D is device specification, S is a testing strategy used to select actions, T is a desired exploration time (e.g. 30 minutes) and I is interval between actions, e.g. in seconds. During each exploratory testing session device resource usage data $U = (\mathbf{u_1}, \mathbf{u_2}, \dots, \mathbf{u_k})$ is collected using a dedicated component. Each $\mathbf{u_i}$ is a time series (vector) of usage values for particular resource (e.g. CPU or RSS). Thus, U is a multivariate time series. The output of testing session is an exploration run $R = (C, U, F)$ which consists of testing configuration C, resource utilization data U and a set of detected failures F. This is depicted in the upper part of the Figure 2.

In the bottom part, an analysis system is depicted. It is responsible for aggregation of test runs and for performing actual detection of anomalous device loads. For each new run app version v' and corresponding run R', system constructs a historical set of similar runs $H = (R_{n_1}, R_{n_2}, \dots, R_{n_k})$. We consider two runs similar if their respective configurations C_i and C_j are the same. Only runs that have successfully passed the regression system tests, e.g. a set of detected failures F is empty, are picked for historical set. H is used to model expected behavior and to compare against it. More regular testing allows to construct better historical sets and thus to model expected resource usage behavior more accurately.

ts	dur	cpu	end_state	thread.name
ts_1	247188	2	S	logd.klogd
261187012418183	12812	2	D	traced_probes0
261187012421099	220000	4	D	kworker/u16:2
261187012430995	72396	2	D	traced_probes1
261187012454537	13958	0	D	traced_probes0
261187012460318	46354	3	S	traced_probes2
261187012468495	10625	0	R	org.schabi.newpipe.HEAD
261187012479120	6459	0	D	traced_probes0
261187012485579	7760	0	R	org.schabi.newpipe.HEAD
261187012493339	34896	0	D	traced_probes0

TABLE I: An example fragment on scheduler data

Nevertheless, we limit the size of H by picking only N latest similar runs to adapt to application evolution. Once anomaly is detected, it may indicate the presence of performance bugs in the new version, hence it is reported to a user for review.

Below we describe resource usage data processing and anomaly detection approach in a greater detail.

B. Collecting resource usage data

The collection of resource usage data takes place during the automated exploratory testing of applications. In our study we focused on Android mobile operating system. Because of that to collect this data we utilize Perfetto [11] tool. We have also considered other options. For example, implementation based on standard Linux tools and using native profilers from Android ecosystem. The first one requires implementation from scratch, while the second one does not provide an API, therefore cannot be used as a part of anomaly detection system.

Perfetto is a cross-platform tracing tool that appeared in relatively new versions of Android (starting with Android 10). It offers a larger number of data sources compared to deprecated predecessor tool Android Systrace and allows to write arbitrarily long files with trace data. It supports collection of different types of resource usage data, including scheduler events, memory RSS, power consumption, etc. At the same time it tries to affect the performance of the system under test as little as possible by recording data in an optimized binary format and by working directly on the device (without intermediate layers, such as Android Debug Bridge).

Within the scope of this work, we have deliberately limited a set of possible resources we track. Collecting data from all available sources at once can lead to a significant overhead during testing and significantly increase storage requirements. Thus, we keep CPU scheduler events from `linux.ftrace` source, which allows one to extract fine-grained CPU usage information, and memory RSS from `linux.process_stats` source. We consider the remaining sources as a part of future work.

The CPU scheduling events are post-processed in order to get CPU usage time series from a collection of scheduler events. Two time series from the scheduling events are extracted: overall CPU usage and CPU usage of target application process. An example slice of scheduler events data is presented in Table I.

According to this table, if we divide the entire data into intervals of fixed length `interval_duration`, then we can understand how much time a particular process was executing during any given interval, and how much the processor was loaded overall in this interval. To convert the received data to a time series, we can split the segment $[\min_{\texttt{ts}\in\texttt{table}} \texttt{ts}, \max_{\texttt{ts}\in\texttt{table}} \texttt{ts}]$ for a certain number of intervals $I_i = [I_i^{\texttt{start}}, I_i^{\texttt{end}}]$ of length `interval_duration` and calculate the sum of the durations of execution of processes p_j that falls within this interval.

After the specified post-processing two one-dimensional series are obtained, which are further combined with memory usage data (RSS) to obtain a three-dimensional time series.

In this paper `interval_duration` was set to 1 second for granularity. This is, on the one hand, a fairly short period of time to track changes in the intensity of resource usage over time, on the other hand — long enough to avoid sharp fluctuations in time series due to too severe partitioning.

C. Anomaly detection

To identify whether target exploration run R is anomalous, we compute the distance between its resource usage time series U and the expected model built on corresponding historical data H. To model expected resource usage behavior we use *barycenters*. Barycenter is a special aggregated representation of a collection of time series, which is obtained by averaging. Time-series averaging is not a straightforward task because small misalignments can cause key features to be lost. Barycenter is good at preserving these features [12]. This specific approach is also useful, because it allows to capture different types of time-series outliers mentioned in II-C, which can be present in resource usage data.

To compare time series and to build a barycenter, it is necessary to define how to calculate the distance between two time series $t_i = (t_i^1, \dots, t_i^m)$ and $t_j = (t_j^1, \dots, t_j^m)$. The trivial approach is to use the Euclidean distance:

$$\|t_i - t_j\|_2 = \sqrt{\sum_{k=1}^{m} \left(t_i^k - t_j^k\right)^2}$$

This method of calculating distances between time series shows poor results because it does not take into account the specifics of time series. In particular, the Euclidean metric is unstable when one of the time series was shifted, stretched or narrowed. For example, if we take time series that are mostly the same, but slightly shifted in time, the Euclidean metric will show a big difference. Among with the occasional minor fluctuations, this pattern is common in resource usage time series captured during automated exploratory testing.

Another distance metric for time series is the Dynamic Time Warping metric (DTW, [13]), which lacks the aforementioned drawbacks of Euclidean distance. In the available notation, the distance between t_i and t_j is defined as follows:

$$\|t_i - t_j\|_{\text{DTW}} = \min_{\pi} \sqrt{\sum_{(x,y)\in\pi} (t_i^x - t_j^y)},$$

where $\pi = [\pi_0, \dots, \pi_K]$ — is a path that satisfies the following criteria:

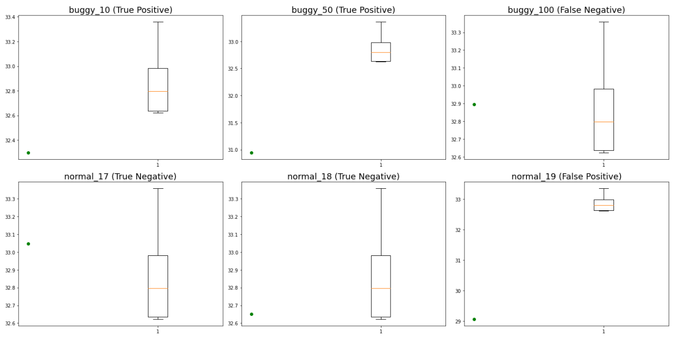

Fig. 3: Example: Distances to barycenter distribution for OmniNotes application

- π — is a set of index pairs $\pi_k = (x, y)$ of time series t_i and t_j, respectively
- $\pi_0 = (0, 0)$, $\pi_K = (n-1, n-1)$
- $\forall k > 0$ $\pi_k = (x_k, y_k)$ and $\pi_{k-1} = (x_{k-1}, y_{k-1})$ are related as $x_{k-1} \le x_k \le x_{k-1} + 1$ and $y_{k-1} \le y_k \le y_{k-1} + 1$

Because of these properties we select DTW as our distance measure between resource usage time series.

For barycenter (expected model) computation we utilize the state of the art method called DTW Barycenter Averaging (DBA) [12]. Following definitions in III-A, given target run R', we extract its resource usage U', exploration history $H = (R_{n_1}, \dots, R_{n_k})$ and corresponding resource usage history in the form of time series $U(H) = \{U_{n_1}, \dots, U_{n_k}\}$. For $U(H)$ we compute its barycenter $B(H)$ using DBA. After obtaining the barycenter, it is possible to calculate the distances between it and each time series in history:

$$D(H) = \{\|u - B(H)\|_{\text{DTW}},\ u \in U(H)\}$$

For detecting whether target exploration run and its resource usage is anomalous we use interquartile range (IQR) method. We compute distance between target resource utilization and barycenter: $d = \|U' - B(H)\|_{\text{DTW}}$. The interquartile range method is based on the first and third quartiles of the set $D(H)$: Q_1 and Q_3 respectively. The *interquartile distance* can be calculated as:

$$\text{IQR} = Q_3 - Q_1$$

We consider the time series U' to be anomalous if its distance d to the barycenter $B(H)$ is greater than $Q_3 + \omega \cdot \text{IQR}$. Here ω is the multiplier, typically known as the "whisker length", that allows to configure the magnitude of acceptable deviation. ω is the hyperparameter and we have tested different values for it (see section IV). Figure 3 shows example for OmniNotes application with boxplots of distances from historical data to the historical data barycenter and a green dot representing prediction for particular resource usage time series.

Since the resource usage of each exploration run is a set of three time series, we have to define when the multidimensional time series is considered as anomaly. We consider resource usage data anomalous when any of its one-dimensional time series is considered anomalous by described method.

IV. Evaluation

To implement our approach we used the automated exploratory testing tool for Android developed at Ivannikov Institute for System Programming of the RAS. We considered three different types of data for evaluation:

- data collected on applications with artificially embedded performance bugs;
- data collected on normal applications, but modified in a special way afterwards;
- data collected on open-source applications with known performance bugs in the old versions.

They are described below in details.

A. Applications with artificially embedded performance bugs

The first set of applications consists of pairs $A = \{(\text{normal}_i, \text{buggy}_i)\}$, where the pair includes application version that has performance issues and an application version without them. To build such set, a self-created set of modified open source applications is considered, where performance problems have been artificially embedded.

We have selected OmniNotes, Forecastie and RedReader open-source applications for this purpose [14]. The taxonomy of performance bugs created in [1] describes complex redundant calculations as one of the most frequent causes of performance issues in real-world applications. Hence, there is a large load on the device's resources, as a result of which the user experience may become worse: target application or UI may lag or become unresponsive until the end of such redundant calculations. To simulate such problems, modified versions of selected open-source applications were created, in which calculations requiring noticeable processing were injected. thus, this part of evaluation dataset allows to evaluate detection of anomalies in CPU usage data.

B. Modified time series

This part of data set was obtained by running exploratory testing on versions of applications without performance bugs with further modification of obtained resource usage data. To simulate real performance bugs related to memory usage, we utilized the results of work [15], where authors analyzed the nature of memory leaks, and adopted their memory utilization classification into two patterns: linearly increasing pattern, saw-tooth pattern (see Figure 4 for example). We have successfully simulated these patterns by modifying the normal resource usage time series for applications Booking.com, Ebay and GnuCash.

C. Old application versions with known performance bugs

In [1] authors have collected a data set consisting of a list of performance bugs in older versions of a set of mobile applications for Android. We planned to utilize it and to evaluate our method on these applications since they are the

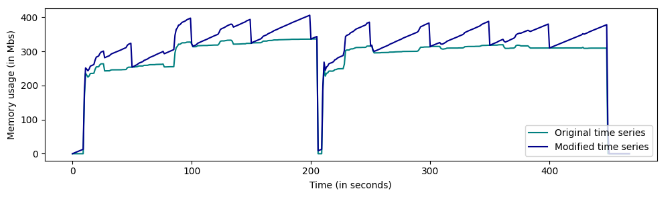

Fig. 4: Example of synthesized anomalous time series

Application name	Number of normal time series	Number of anomalous time series	Method of obtaining anomalous time series
Booking.com	31	19	Modified time series
Ebay	31	19	
GnuCash	31	19	
OmniNotes	20	8	Artificially embedded performance bugs
Forecastie	20	8	
RedReader	20	8	

TABLE II: Description of generated data set

closest to real-world examples of performance bugs. However, we faced two major issues. First, there were problems with building these versions of applications which were mainly related to outdated dependencies or build systems. Second, we were not able to easily reproduce specific performance problems for a subset of successfully built applications using exploratory testing in a reasonable amount of time. Because of these reasons we had to give up using this specific dataset in the scope of this paper. We are leaving the task of creating a similar dataset for a set more modern Android applications for the future work.

D. Evaluation methodology

In the end, our implementation helped us obtain a data set which contains approximately 60 hours of exploratory testing. For each application $A \in$ (OmniNotes, Forecastie, RedReader, Booking.com, Ebay) we have a set of time series $T_A = \{t_{\text{normal}_1}, \dots, t_{\text{normal}_{k_A}}, t_{\text{anomalous}_1}, \dots, t_{\text{anomalous}_{m_A}}\}$, represented in table II. The historical subset of fixed size 12 was chosen randomly from the normal time series. Every other time series from T_A was treated as a target time series for anomaly detection using selected historical set. It can be observed experimentally that the metrics can change depending on the choice of historical set, so we decided to calculate metrics for 30 random historical sets and then average the results. We also wanted to check the impact of ω (whisker length) parameter on the result (see section III-C). The total number of inputs for anomaly detection for every w is

$$\sum_{A} \left[(k_A + m_A - \texttt{hist\_size}) \cdot \texttt{iterations} \cdot p_a \right] = 6300,$$

where $\texttt{hist\_size} = 12$, $\texttt{iterations} = 30$ and p_A represents the number of 1-dimensional time series which can be anomalous: $p_A = 1$ for the first 3 apps from the table II and $p_A = 1$ for other 3 apps, since our resource usage data

	TP	TN	FP	FN	Precision	Recall	F_1-score
$\omega = 0.0$	2500	1669	1481	650	0.628	0.794	**0.701**
$\omega = 0.5$	2169	2094	1056	981	0.673	0.689	0.680
$\omega = 1.0$	1979	2396	754	1171	0.724	0.628	0.672
$\omega = 1.5$	1688	2550	600	1462	0.738	0.536	0.621

TABLE III: Evaluation results

types are CPU usage per target process, overall CPU usage and memory usage per target process.

To obtain a quantitative assessment of the quality of anomaly detection, the metrics Precision, Recall, F_1-score were calculated:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$F_1\text{-score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

E. Results and future work

The evaluation results are presented in table III. The values of the first 4 columns represent the numbers of 1-dimensional time series on which anomaly detection was performed. For these applications, the 0.0 is the most optimal value for ω with F_1-score $= 0.701$. With an increase of the w parameter, Recall decreases and Precision increases. We believe that these metrics can be improved, e.g. by designing additional predicates that will work with a specific resource usage type and utilizing its specifics (e.g. some heuristics for memory usage). Also, we consider localization of anomalous time series intervals as a part of future work.

V. Related work

This section provides a short overview of existing approaches that are close to the topic of this paper. The overview includes two research directions: identification of non-functional bugs via dynamic testing and time-series data anomaly detection.

Anomaly detection in multidimensional time series is a popular research area. Many methods have been proposed for different purposes including detection of anomalous resource usage in cloud-based computer systems. In [16], [17], the task of detecting anomalies in multidimensional time series in real-time is set. Both of this papers use historical data, but they are aimed at real-time anomaly detection, which is different from our goal.

The work [18] describes anomaly detection in multidimensional time series. Signature matrices are constructed for different time series intervals, which are then used by a CNN to find anomalous areas. However, the approach proposed in this article is used to find anomalies within a single multidimensional series without any historical data set, so this method differs from ours.

The works [7], [8] describe resource usage data anomaly detection methods. However, both of them use supervised learning and do not imply any usage of historical data for a particular resource usage time series.

Paper by Gomez et al. [19] describes an approach to detect performance regressions which is similar to ours. This approach also collects resource usage data and performance metrics. Contrary to our approach with barycenters and DTW it uses IQR method on distribution statistics and thus does not consider time series specifics.

Another work by Jindal et al. [15] suggests a technique to detect memory related issues and memory leaks in particular. The detection is performed in the context of one run only and does not consider changes in memory usage patterns between different versions of software.

In the article [1], a classification of performance bugs was carried out and their survivability (time taken to detect and fix) was investigated. To find examples of such bugs, the authors searched by keywords for open source versions of applications in which performance bugs were fixed, and then manually classified each fixed error. The authors of the work posted the resulting data set of 500 versions of applications with classified types of bugs.

As a result of a review of existing solutions, only one data set suitable for the tasks of this work was found: the authors of the article [1] provided a collected classification of performance bugs in applications on iOS and Android, which is a set of application versions (corresponding commits) in which the bugs were fixed. We tried to use it during evaluation in IV.

VI. Conclusion

In this paper, we've presented a novel approach to detect anomalous device loads and hence potential performance issues. Our approach is integrated with exploratory testing framework and uses information about previously executed test runs to build the expected resource usage model in the form of time series barycenter with DTW as a distance metric. This choice allows us to detect different types of time series anomalies, including point outliers, sub-sequence outliers and time-series outlier as a whole. We have integrated our approach with exploratory testing tool for Android and empirically evaluated it on a set of real-world applications with injected performance issues. Our results show that the suggested approach can be successfully applied to detect anomalous device resource usage and potential performance regressions with F1-score equal to 0.701.

VII. Acknowledgment

We are grateful to Alexander Spiridonov, Mikhail Ermakov and Georgiy Pankratenko for assistance with some of tool developments and useful comments.

References

[1] A. Mazuera-Rozo, C. Trubiani, M. Linares-Vásquez, and G. Bavota, "Investigating types and survivability of performance bugs in mobile apps," *Empirical Software Engineering*, vol. 25, pp. 1–43, 05 2020.

[2] A. Nistor, T. Jiang, and L. Tan, "Discovering, reporting, and fixing performance bugs," 05 2013, pp. 237–246.

[3] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 23–26.

[4] ——, "Humanoid: A deep learning-based approach to automated black-box android app testing," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1070–1073.

[5] J. Eskonen, J. Kahles, and J. Reijonen, "Automating gui testing with image-based deep reinforcement learning," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, 2020, pp. 160–167.

[6] W. Guo, L. Shen, T. Su, X. Peng, and W. Xie, "Improving automated gui exploration of android apps via static dependency analysis," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 557–568.

[7] V. Vladareanu, V. Voiculescu, V.-A. Grosu, L. Vladareanu, A.-M. Travediu, H. Yan, H. Wang, and L. Ruse, "Detection of anomalous behavior in modern smartphones using software sensor-based data," *Sensors*, vol. 20, 05 2020.

[8] L. Gheorghe, B. Marin, G. Gibson, L. Mogosanu, R. Deaconescu, V. Voiculescu, and M. Carabas, "Smart malware detection on android," *Security and Communication Networks*, vol. 8, pp. n/a–n/a, 09 2015.

[9] X. Zhang, F. Meng, P. Chen, and J. Xu, "Taskinsight: A fine-grained performance anomaly detection and problem locating system," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, 2016, pp. 917–920.

[10] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?" 2015.

[11] "Perfetto - system profiling, app tracing and trace analysis," https://perfetto.dev/.

[12] F. Petitjean, A. Ketterlin, and P. Gançarski, "A global averaging method for dynamic time warping, with applications to clustering," *Pattern Recognition*, vol. 44, no. 3, pp. 678–693, 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S003132031000453X

[13] S. Z. Li and A. Jain, Eds., *Dynamic Time Warping (DTW)*. Boston, MA: Springer US, 2009, pp. 231–231. [Online]. Available: https://doi.org/10.1007/978-0-387-73003-5_768

[14] "pcqpcq/open-source-android-apps: Open-source android apps," https://github.com/pcqpcq/open-source-android-apps.

[15] A. Jindal, P. Staab, P. Kulkarni, J. Cardoso, M. Gerndt, and V. Podolskiy, "Memory leak detection algorithms in the cloud-based infrastructure," 06 2021.

[16] P. Filonov, F. Kitashov, and A. Lavrentyev, "Rnn-based early cyber-attack detection for the tennessee eastman process," 2017.

[17] C. Wang, K. Viswanathan, L. Choudur, V. Talwar, W. Satterfield, and K. Schwan, "Statistical techniques for online anomaly detection in data centers," in *12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops*, 2011, pp. 385–392.

[18] C. Zhang, D. Song, Y. Chen, X. Feng, C. Lumezanu, W. Cheng, J. Ni, B. Zong, H. Chen, and N. Chawla, "A deep neural network for unsupervised anomaly detection and diagnosis in multivariate time series data," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 1409–1416, 07 2019.

[19] M. Gomez, R. Rouvoy, B. Adams, and L. Seinturier, "Mining test repositories for automatic detection of ui performance regressions in android apps," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, 2016, pp. 13–24.