IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 21 September 2022, accepted 7 October 2022, date of publication 14 October 2022, date of current version 20 October 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3214971

APPLIED RESEARCH

Method for Profile-Guided Optimization of Android Applications Using Random Forest

ANDREI VISOCHAN[1], **ANDREY STROGANOV**[1], **IVAN TITARENKO**[1], **SERGEI LONCHAKOV**[1], **STANISLAV MOLOGIN**[1], **SVETLANA PAVLOVA**[2], **ANASTASIA LYUPA**[3], **AND ANNA KOZLOVA**[4]

[1]Platform Laboratory, Samsung Research Russia, 127018 Moscow, Russia
[2]Solution Development Laboratory, Samsung Research Russia, 127018 Moscow, Russia
[3]Moscow Software OS Laboratory, Huawei Technologies Russia, 121614 Moscow, Russia
[4]Kelly Services CIS, 129090 Moscow, Russia

Corresponding author: Andrei Visochan (a.visochan@samsung.com)

ABSTRACT When choosing a smartphone, many users are guided by the performance of smartphones and the speed of applications. Because it is difficult to measure the application speed directly, the speed of the application startup is considered and used for comparison. Android runtime (ART) uses several technologies to speed up applications. The first approach is the just-in-time (JIT) compilation of frequently used methods at runtime. The second approach is to compile entire application code ahead of time (AOT). The performance profile is a way to strike a balance between JIT and AOT. The runtime optimization features were introduced in Android Nougat in form of profile-guided optimization (PGO). By aggregating data from a multiplicity of users and devices in Play Store, ART profiling significantly speeds up this process and makes its outcome available to all users alike. We propose a machine learning based method called SPMLGen for generating application profiles used in the optimization. We avoid time delays caused by the need to collect information in advance to perform optimization and ensure user privacy. With profiles generated by SPMLGen, we obtain approximately the same application launch time as with profiles from Play Store. Measurements were taken on Samsung Galaxy S22 and A52 devices with Android 12 firmware and several dozen Samsung applications.

INDEX TERMS Android, application launch time, classification, machine learning, probability tree, profile-guided optimization, random forest.

I. INTRODUCTION

Program performance (speed, memory usage, network resources, power consumption, etc.) is crucial for users' satisfaction with their smartphones. Users want mobile applications to work quickly and smoothly, and to consume as little as possible of the memory and power of the user's electronic device. Poor program performance can negatively affect the performance of the electronic device itself, thus degrading the device rating and its sales.

Program code optimization was proposed to improve program performance. Such optimization can be performed manually or automatically, for example, by a compiler or other tools. Compiler optimization is the step of compiling a program that generates code that is more efficient (faster to execute, shorter, results in less power consumption by the device on which it is executed, etc.) than the original code. One of the compiler optimization methods is profile-guided optimization (PGO). Profiling is a method of code analysis that measures performance characteristics such as the frequency and duration of function calls, memory used, etc. In the process of PGO, based on the existing profile of the program, a part of the program is selected as important and aggressively optimized, possibly at the expense of another, less important part of the program [1]. Profiling can generally be divided into static (offline) and dynamic (during the execution). Dynamic profiling can use instrumentation, hardware counters [2], electromagnetic emanations [3], and program

The associate editor coordinating the review of this manuscript and approving it for publication was Sotirios Goudos.

pre-runs, possibly in an emulator. Dynamic profiling has the following disadvantages: it by definition requires code to run, negatively affects program execution performance, requires representative data, and is time consuming. Static profiling can use abstract interpretation [4], heuristics (e.g., to predict transitions [5], [6], [7]), Markov processes (e.g., to estimate the relative frequency of execution of basic blocks [8]), data flow analysis (e.g., to determine the range of values of constants, numeric variables [9], [10]), traditional machine learning (ML) algorithms (e.g., to predict the critical path [11], detect parallelism [12]), deep learning (e.g., to predict the critical path [13], estimate the processing speed of basic blocks [14]), or reuse a legacy (old) profile [15].

For a user to benefit from profile-guided optimization, a profile must be available on the user device prior to running the program. Profiles can be prepared and delivered from cloud storage (e.g., profiles from Google Play Store storage for Android applications). When an application is used by a plurality of users and there are many profiles in the storage, the profile can be available and provided at the time of loading. In this case, the application can be compiled with the profile and optimized from the start to obtain better performance. Although improved performance is achieved, this approach still has drawbacks. An application must be launched to generate profiles. Consequently, profiles may not be available for new applications or new versions of existing applications. Additionally, to generate the program profile in cloud storage, it is necessary to transfer data about the execution of said program on the user device, as well as, possibly, data about the user device or even about the user themselves (e.g., in the form of a user/program profile), which is not desirable from the standpoint of privacy. Finally, the aggregated profile may not be suitable for a particular user or the electronic device of the user. In the application storage, a profile for an application may not be available if there is a completely new application, a new version of a legacy application, if not enough users have used the application, or not enough users have used a particular device model, architecture, language region, and so on. In this case, as an alternative, a profile can be generated on the device during the execution of a particular application. Thus, the application is downloaded from the storage without a profile, the profile is then generated on the user's electronic device after the application has been used for some time, and then the generated profile is used for optimization. This profile may be sent to storage for aggregation. Although some aspects of program execution can be improved using this approach, it still has the following disadvantages. Because the profile is not available when the application is first launched, the application is not optimized during the first launch, and the performance of the application may be reduced during this period. In addition, a profile is generated based only on the user data. Thus, such a profile may not reflect the various stages of an application's execution, and when the user encounters these stages, the performance of the application may not be optimal.

We make the following three contributions.

1) We propose a method called SPMLGen for generating a program profile based on machine learning for the PGO of an Android application.
2) We train device-specific models for Samsung devices and present an analysis of the predictive quality of these models, which is measured using common metrics.
3) We analyze the performance of SPMLGen using sets of Android application packages (APK). We perform PGO of the application using the generated program profile and compare it with PGO using Play Store profile. Extensive experimental results demonstrate the effectiveness of SPMLGen. It achieves almost the same average application startup acceleration compared to Play Store for applications similar to the ones SPMLGen model was trained on.

The remainder of this paper is organized as follows. In the next two sections, we provide an overview of the fundamentals of compiler optimization (Section II) and PGO techniques in Android (Section III). Then, in Section IV, we provide an overview of related work, while in Section V, we present our method. In Section VI the experiment setup, methodology and results are described. The limitations are discussed in Section VII, and the conclusions and possible future work are presented in the final section.

II. BACKGROUND

Most Android applications are written in Java, which is designed as a general-purpose programming language that allows programmers to write once and run anywhere. Typically, Java applications are compiled into an architecture-independent bytecode that can run on any processor for which runtime implementation exists, such as a virtual machine. Android has gone further and has its own bytecode format called Dalvik Executable (DEX) bytecode and its own runtime called ART. It can compile DEX bytecode in several ways, providing additional options for applications and system services performance for a particular device.

ART includes a just-in-time (JIT) compiler [16] with code profiling that continually improves the performance of Android applications as they run. The JIT compiler complements ART's current ahead-of-time (AOT) compiler, improves runtime performance, saves storage space, and speeds up application and system updates. It also improves the AOT compiler by avoiding system slowdown during automatic application updates or recompilation of applications during over-the-air (OTA) updates [16].

Application methods can be in three different states:

1) Interpreted (DEX code, ".dex" file), leading to slow execution.
2) JIT compiled, which is faster than the interpreted state, but compilation needs to be performed each time the application starts.

3) AOT compiled (".oat" file), which results in the fastest execution, but requires more storage space and some preparation before running the application.

Android provides the ability to compile methods selectively using ART optimizing profiles (PGO) during AOT compilation.

Steps for AOT compiling using ART optimization profiles [17]:

1) The user runs the app, which then triggers ART to load the .dex file. If the ".oat" file is available, ART uses it directly. Although ".oat" files are generated regularly, they do not always contain a compiled code (AOT binary). If the ".oat" file does not contain compiled code, ART runs through JIT and the interpreter to execute the ".dex" file. JIT is enabled for any application that is not compiled according to the speed compilation filter (i.e., "compile as much as you can from the app").
2) The JIT profile data is dumped to a file in a system directory.
3) The AOT compilation daemon (dex2oat) parses that file to drive its compilation.

Google reported that, on average, apps' cold starts are at least 15% faster across a variety of devices when profiles are available. In some cases, startup is even 40% faster [17].

Remarkably, on average, ART profiles contain approximately 20% of all application methods (even less if we consider the actual size of the code). For some apps, the profile covers only 2% of the code, and for others, the value reaches 60% [18].

Thus, ART profiles allow Android to achieve a significant speed-up of application launch time with minimal storage space and CPU time for compilation.

III. PROFILE GUIDED OPTIMIZATION

The previous section presents background information to assist in understanding the following PGO approaches. In this section, we describe existing approaches that use PGO to improve applications performance. The first approach, *background DEX optimization*, is built into ART and is available for almost any Android device. This feature performs profile-based compilation in the background while the device is idle and charging. JIT collects the methods and classes that are frequently used (i.e., hot methods and hot classes) during application execution. When the collection time threshold is exceeded, ART saves hot classes and methods (i.e., profile info) to a file (profile file). In this step, ART has a profile file for the application and is ready to apply PGO. This is a common method for Android devices to use PGO. The above solution has the following disadvantages related to profile generation and usage:

1) JIT requires a separate thread to collect information regarding the executed methods, which additionally loads the system as a whole.
2) The profile is generated after using the application. Background dexopt (i.e., applying PGO based on the generated profile) optimizes apps in the idle maintenance mode. It could be a few days before a user perceives benefits [18].

IV. RELATED WORK

In Android Pie, Google introduced ART optimizing profiles in Play Store, a new optimization feature that significantly improves the application startup time after a new installation or update [18]. The main idea of this approach is to collect profiles from user devices and aggregate them into a profile on the Cloud. This is possible because applications usually have many commonly used code paths (hot code) between a multitude of users and devices, for example, classes used during startup or critical user paths [18]. This approach is shown in Fig. 1 and consists of three steps.

A. COLLECTING PROFILES

Initial profiles are generated on the devices that first received an application update. These profiles are produced by ART, as described at the beginning of this section. Finally, the profiles are uploaded to Play Store.

B. AGGREGATING PROFILES

The collected profiles are aggregated into an average (core) profile. This profile contains only anonymous data about the code that is frequently observed across a random sample of sessions per device [18].

C. INSTALLING PROFILES

Play Store delivers a core profile to a user device with an APK file. The delivered profile is managed exclusively by the Android platform and is applied when installing the application. The expected increase in the application launch speed after compiling, based on the profile, is shown in Fig. 2.

This solution has the following problems related to profile generation and profile use.

1) Profiles are not available outside the Google Play Store, which reduces application coverage.
2) To generate a core profile, users must participate in the initial profile generation on their devices.
3) An aggregated core profile generated for different users and devices is averaged. Thus, the profile may not account for differences between individual user devices, environments, and scenarios, that is, it may not be identical to a user pattern.
4) User privacy. The collection and aggregation of user profiles require the transfer of user data and other data from the user's device, and the user must consent to this transfer.
5) Maintenance of cloud profile storage requires significant resources.

As an alternative to the profiles from Play Store, in the past, we implemented two different approaches to PGO. The first is the profile reuse tool described in [15] and [19]. It operates before a downloaded application is installed if a profile is not delivered during installation. The tool compares a previous

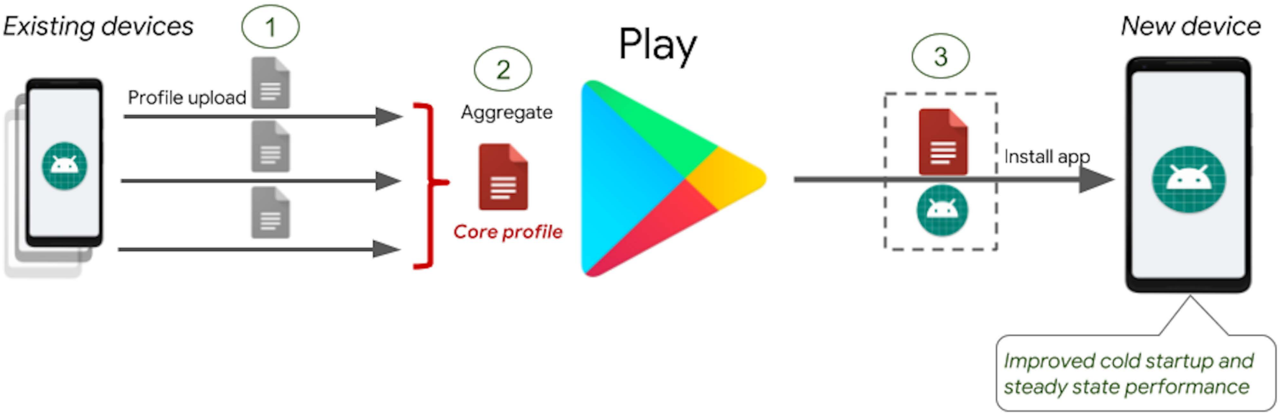

FIGURE 1. Cloud profile aggregating and delivering to apply PGO at application installation [18] .

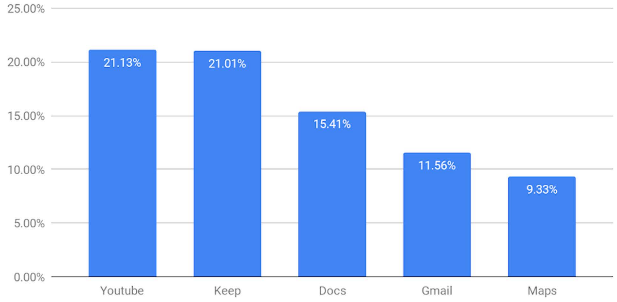

FIGURE 2. Applications startup time speed-up [18].

profile (old profile), which is based on a user's usage pattern, and a bytecode that was previously stored in an application (old APK) with a new bytecode that is stored in a new application (new APK). The tool groups information about classes and methods from an old APK based on a profile and information from a new APK into metadata.

Metadata represents information about a class or method. It includes information such as the method return type, method arguments, class information, number of packages in the method return type, whether the method return type is an object type or a primitive type, and whether an array is used as a return type. The same information is grouped for a method's arguments, if they are present. Finally, it contains information regarding the method's body (e.g., command code (opcode)) [19]. Based on the invariant representation of methods and classes of the old profile, and methods and classes in the new APK, the tool produces hash values. Then, it uses the generated hashes to perform comparison (matching) to determine whether the methods or classes are identical. The tool produces a new profile based on the matched methods and classes. The profile is then used for compilation with PGO.

The main advantages of this approach are the absence of cloud infrastructure, the absence of the need to install an application from Play Store only, and the tool running directly on the device. However, this approach has several disadvantages.

1) The approach of reusing legacy profiles is applicable to future versions of programs only when legacy versions are installed on a device and profiles are collected for the versions.
2) Sometimes profiles obtained from previous versions of applications cannot be used in this approach for several reasons (e.g., profile size limit, number of classes and methods limit).
3) Analyzing applications, making comparisons, and creating new profiles can negatively affect the installation times of applications.
4) Profiles based on static heuristics may be inaccurate.
5) Using legacy profiles for PGO may result in worse performance than using newly generated profiles. The less similar the new version of the application is to the legacy version, the worse the effect of PGO will be with the legacy profile.

However, profiles can be generated based on application execution during installation. This approach is described in [20]. This method is based on application pre-execution (e.g., running it in the background) to obtain a profile generated by ART according to the executed methods. If a profile related to the application is not received during installation (i.e., it is not downloaded with the application via Google Play), the application is executed in a background state, so the user does not notice the execution of the application. The method uses a virtual display to execute an application to prevent the application from being displayed on the main display of the device. The execution is limited (i.e., performed in a sandbox), and some functions, such as communication, sound, notification, and log output functions [20] are not available for the application. A profile that is generated based on the execution is used during PGO. A primary goal of this method is to generate profiles for applications that do not have it at installation, thereby improving application performance and decreasing its launch time by applying PGO.

The described approach has the following disadvantages:

1) Program pre-execution in the background is resource-intensive and can lead to delays and visual effects disturbing the user when using the device.

2) The resulting profile may not include classes and methods that can be a hot part of the application code because of the limitations of execution in a sandbox.
3) These profiles may not reflect the behavior of the program when data are entered by a real user.

V. METHOD

Several ML algorithms are commonly used for binary classification problems, such as deep neural networks [21], ensemble models [22], and others. All models have different tunable parameters and have their pros and cons, such as model size, sensitivity to data noise, training, and prediction time, etc.

The ensemble models demonstrate good results [22], [23], [24] when learning new patterns of features is necessary. An ensemble model is a collection of classifiers (or regressors), where each classifier was trained on a portion of dataset (the subset of samples and features).

A. FEATURES

To train the ML model to predict methods and classes hotness (sometimes we will not distinguish methods and classes and use term "samples" for referring to them) in APK file, we must define a set of features — pieces of information about a method or a class, which, in some combinations, may be common for hot samples. A list of hot samples is provided in an ART profile. During the training, we add to each sample a label indicating the presence of this method or class in the reference profile. The list of samples with their modifiers (e.g., `private`, `public`, `final`), arguments, other information, and bytecode can be found in a DEX file dump. To train an ML model, we must process the dump of each sample into a set of features.

Features are specific properties of a class or method, such as the length of the name, number of method parameters, its modifiers, return type. A feature set defines the quality of prediction. Although each feature may have a small correlation with the target value, the sets of features are proven to be useful for prediction. We propose a set of features combining two different sets: common and token features. Common features may be binary or numerical, and they contain general information about method or class. For instance, binary features represent method or class modifiers (whether a method is `native`, `synchronized`, `static`, `void`, etc.). Numerical features are the length of the name, number of method parameters, minimal, maximal, and average length of method argument names, and so on. Another interesting example of a binary feature is class hotness. For the methods, this feature provides information about hotness of the parent class. First, we run predictions for classes, and after hot classes are found, we modify the dataset by setting the predicted class hotness feature values. Although classes and methods may have various specific features (i.e., method return type or number of methods in class), we let methods inherit features from their parent class and aggregate method-specific features into class features.

```
<method name="SomeMethod" return="[Ljava/lang/Object;"
abstract="false" native="false" synchronized="false"
static="false" final="true" visibility="public">
<parameter name="arg0" type="I"> </parameter>

<code><registers>2</registers> <ins>2</ins>
<outs>0</outs> <insns_size>3</insns_size>

<bytecode>
19c0fc: |[19c0fc] Landroid/support/v4/media/
session/SomeClass$SomeAction$1;
.SomeMethod:(I)[Ljava/lang/Object;
...
</bytecode> </code> </method>
```

FIGURE 3. DEX dump of a method.

TABLE 1. Example of a dataset.

isClass	numMethodsInClass	abstract	...	Ljava	Lorg	In profile
1	3	1	...	1	2	0
0	2	1	...	0	1	1
0	2	0	...	1	1	0
...	...	...	...	1	...	...
1	7	0	...	4	3	1

Token features are generated by extracting tokens (maximal sequences of alphabetic characters) from function signatures and bytecode. In Fig. 3 we see that return type of the method is "`[Ljava/lang/Object;`". Here "Ljava", "lang" and "Object" are the tokens. During the training, we count the occurrences of each token in the dataset and select the most frequent tokens. Each of these tokens defines a new feature: the number of occurrences of a particular token in a class or function dump. By combining common and tokenized features, we define the feature space for one APK file (see Table 1), which usually contains $\sim 10^5$ samples.

B. CLASSIFICATION MODELS

Our model is based on Random Forest (RF) because of its moderate set of parameters with predictable influence on quality, speed, and model size [29]. Random Forest is an ensemble model in which the classifiers are independent decision trees, also known as probability trees. Each node of such a tree corresponds to a feature and has a "splitting" value [30]. It splits the dataset into two subsets with respect to the node feature. The samples with feature value less than or equal to the splitting value are put into one dataset and the rest of the samples are put into the other (see Fig. 4). The tree is traversed such that the dataset is partitioned until the maximal allowed tree depth is reached or the datasets in each leaf contain only samples of the same class. Each leaf of such tree holds probability of the sample to be classified as "hot".

To train decision trees we use two techniques:

1) Bootstrap aggregating [31], also known as "bagging", is a method for creating new datasets from the original. Each decision tree in the ensemble has its own dataset with the number of samples equal to the

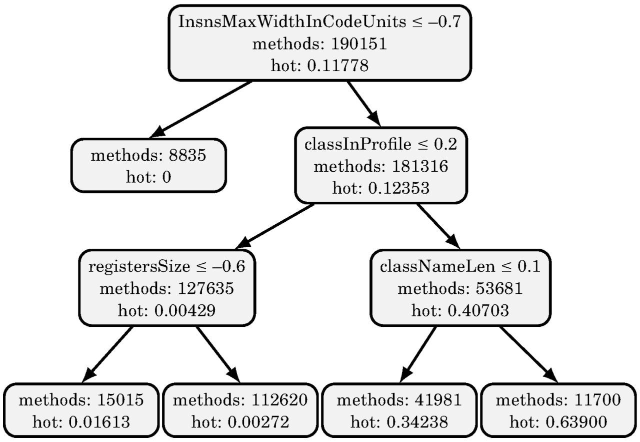

FIGURE 4. Decision tree of height 3. For each node we see the number of available samples in dataset, and a probability of sample to be hot (rounded up to 5 digits).

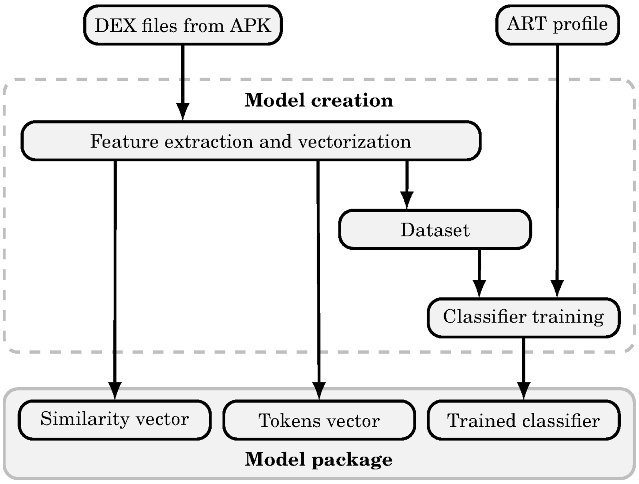

FIGURE 5. Creation of a model.

original dataset. Trees are built by sampling with replacement from the original dataset (some samples may be included several times, whereas others are not included at all). When the bagged dataset is the same size as the original dataset, it contains approximately $1 - \frac{1}{e} \approx 63.21\%$ of unique training samples [32].

2) Feature subspacing [33] defines a subset of the features available for training for each node of the decision tree. For a particular node, we randomly select $[\log_2(\text{number\_of\_features})] + 1$ of all features and use them to split the dataset.

The feature corresponding to the node and its splitting value are selected by maximizing the information gain [34], which can be roughly explained as the amount of information about the sample hotness a feature provides.

C. ENSEMBLE OF MODELS

Hot samples prediction quality is essential. To improve it, we train several models on different APK files and pack them together, creating an ensemble of models. When predicting hot samples for APK, we aim to use the model that provides the most accurate prediction across all models in the ensemble. The ensemble contains the vector of the most common tokens found in all APK files used for training models and a set of model packages.

Each RF model is trained using its APK file. During training, we create a tokens vector and a similarity vector and pack them into a model package (see Fig. 5). The tokens vector holds the most common tokens that define a set of features. The similarity vector is used to select the most suitable model from the ensemble for a given APK. It is created using the following algorithm:

1) Let $\mathbf{T} = (t_1, \ldots, t_n)$ be a vector of the most common tokens in the ensemble of models.
2) Create vector $\mathbf{S} = (s_1, \ldots, s_n)$, where s_i denotes the number of occurrences of token t_i in the token set of this model. S is the similarity vector.

Thus, an ensemble of models contains the vector of the most common tokens over all the APK files used for training and a set of model packages.

To select the most suitable model for prediction, we use cosine similarity, which is defined as the cosine of the angle between two similarity vectors. Let a and b be n-dimensional vectors and θ be the angle between them. Then

$$\text{cosine similarity} = \cos(\theta) = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} a_i^2} \cdot \sqrt{\sum_{i=1}^{n} b_i^2}}$$

Note that maximal similarity is 1, which corresponds to equal vectors up to scaling.

When predicting the hot samples in a given APK file, we create a similarity vector $\mathbf{V}$ for this APK using the vector of the most common tokens of the ensemble, just like when we trained the models. We then find a model with a similarity vector that maximizes the cosine similarity with $\mathbf{V}$.

After a model for inference is selected, we create a dataset for a given APK file by extracting common features and token features, which are defined by the model vector of tokens. This dataset is provided to the RF model, which calculates the predictions (see Fig. 6).

D. INFERENCE QUALITY

Several metrics are commonly used to compare ML models and measure prediction accuracy (see [25], [26], [27], and [28]), such as F1-score, Brier scoring, ROC, and PR AUC, etc. Performance metrics based on the F1-score [25] generally provide accurate estimation for unbalanced problems and are easy to compute. Let TP (true positive) be the number of correctly predicted hot samples, FP (false positive) be the number of cold samples predicted as hot, TN (true negative) be the number of correctly predicted cold samples, and FN (false negative) be the number of hot samples predicted as cold. Precision is defined as the fraction of correctly predicted hot samples to all samples that are predicted to be hot, and Recall is the fraction of correctly predicted hot samples of all

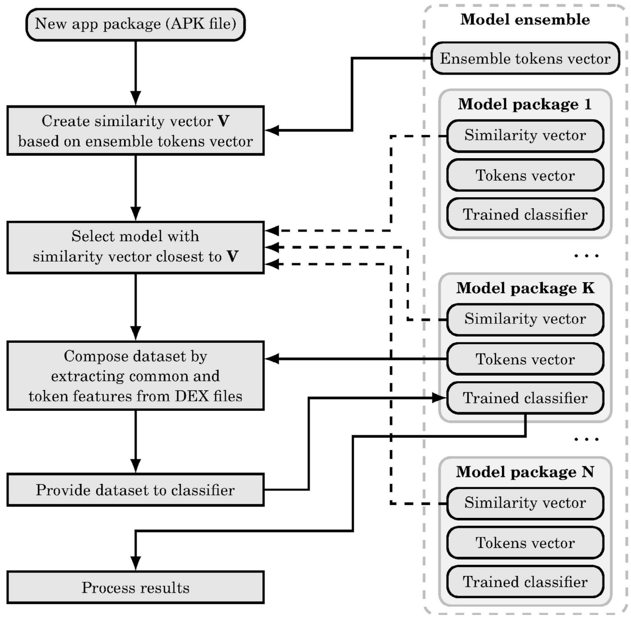

FIGURE 6. Inference using ensemble of models. Here we suppose that Model package K is most suitable for inference.

hot samples:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{1}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{2}$$

F1-score is defined as harmonic mean of precision and recall:

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{3}$$

We will use F1-score to estimate the quality of predictions and to tune model parameters.

E. THRESHOLD

As a result of the inference, we obtain the probability of being hot for each sample. To distinguish hot and cold samples, we have to define a threshold — a probability value for which all samples with greater probabilities are considered hot and the rest are cold.

Note that when the threshold changes, the F1-score also changes, with a maximum possible value equal to 1 when the precision and recall are equal to 1. To find the threshold value that maximizes the F1-score, we require a dataset for which hot samples are known. Thus after inference, we have a vector of probabilities and reference values. Note that the probabilities are real values in segment [0, 1], and the reference values are exactly 0 (for a cold sample) or 1 (for a hot sample).

We use a dynamic programming approach to find the threshold that maximizes the F1-score. Consider a sorted vector $\mathbf{Q}$, each element of which is a pair of a probability and the corresponding reference value (p_i, r_i). The pairs are ordered by increasing of probability:

$$\mathbf{Q} = \langle (p_1, r_1), (p_2, r_2), \ldots, (p_n, r_n) \rangle ,$$

where $p_i \leq p_j$ for all $i < j$.

Due to the ordering, the values of the F1-score for each probability p_i used as the threshold can be effectively computed in a single pass over vector $\mathbf{Q}$. We look for a probability that maximizes the F1-score. Let us iterate over $\mathbf{Q}$ and sequentially compute the F1-score for a threshold equal to each p_i. The initial threshold value is 0, and all samples are classified as hot, that is, TP is equal to the number of hot samples in the dataset, FP is equal to the number of cold samples, and FN $=$ TN $=$ 0, which means that the F1-score is equal to 0. Denote F as the maximum F1-score value currently found and T is the threshold at which F is computed. Initially $F = 0$ and $T = 0$ as we start with the lowest probability value.

Suppose TN, FP, TP, FN are known for $(i-1)$-th iteration. Then, on i-th iteration, there are two possible outcomes:

1) if r_i equals 1, then for a threshold equal to p_i the sample with hotness probability p_i is considered cold, but the reference label is 1 (hot); therefore, we decrease TP counter and increase FN counter:

$$\text{TP} := \text{TP} - 1,\ \ \text{FN} := \text{FN} + 1,$$

2) if r_i equals 0, then the hotness of the sample is predicted correctly as cold; therefore, we decrease FP counter and increase TN:

$$\text{FP} := \text{FP} - 1,\ \ \text{TN} := \text{TN} + 1.$$

Knowing these new values, we compute the F1-score, and if it is greater than F, we update F with the current F1-score and T with p_i. This approach allows effective computation of the threshold value T which maximizes the F1-score. This allows us to compute the best threshold values for a set of APK files. Our experiments showed that the most appropriate average threshold value for our datasets is 0.1.

VI. RESULTS

We downloaded different versions of Samsung applications, selected 47 packages, and created four groups:

1) 28 applications on which we trained our model,
2) a newer version of each application from Group 1,
3) a newer version of each application from Group 2,
4) 19 applications that were not used in the first group.

We used two Samsung devices in our experiments, Galaxy S22 and A52. To train a device-specific model, we used device profiles, that is, profiles obtained from a 10-seconds application launch on that device. The models trained using profiles generated on Galaxy S22 and A52 exhibited similar prediction quality and other properties.

A. PREDICTION QUALITY

During the experiment, precision, recall, and F1-score (see formulas (1) – (3)) were obtained for each APK file in each group. Prediction quality was measured using the F1-score and PR AUC metrics [35].

Our experiments show that the F1-score distribution for applications from groups 1 to 4 degrades as the group number

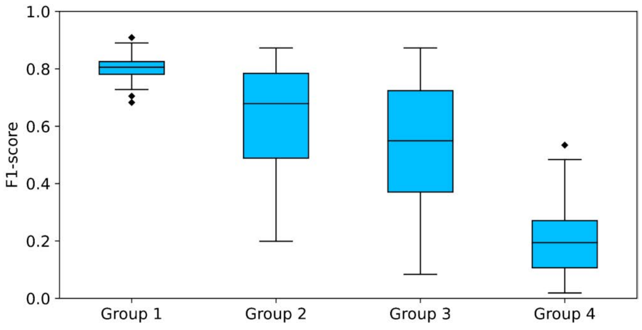

FIGURE 7. F1-score distribution for different APK groups.

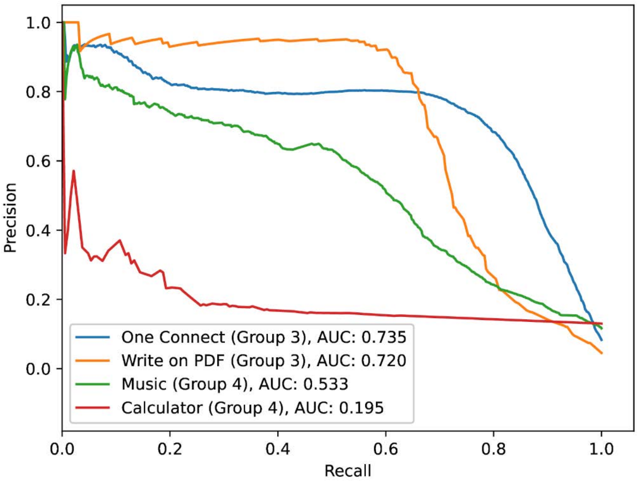

FIGURE 8. PR curves for applications from Group 3 and Group 4.

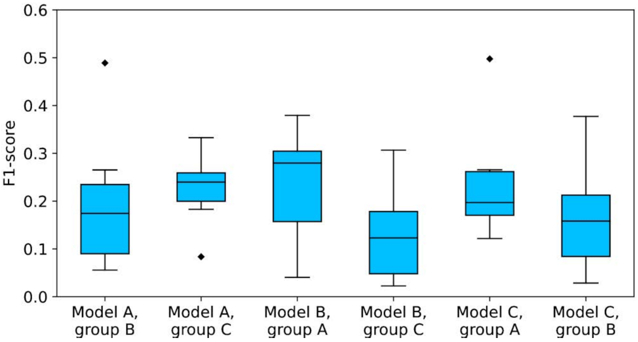

FIGURE 9. F1-score distribution for predictions of unfamiliar applications using models A, B and C.

increases. This means that the prediction quality for applications decreases as newer versions of applications are released, tending to the quality of prediction for applications that are completely unknown to the model (Group 4). On Fig. 7 we see a box plot depicting the changes in the F1-score from group 1 to group 4. Measurements were performed using a threshold value of 0.1. Horizontal segments above and below the boxes are the maximum and minimum values, respectively, and all F1-score points are distributed between them, excluding the outliers — a small number of data points that differ significantly from other values. The range between the minimum value and the bottom of the box contains the lowest 25% of data points, as well as the range between the top of the box and the maximum value. Thus, the range between the bottom and top of the box is 50% of all values. The horizontal line in a box represents the median, which is a value that separates the lower half of all the points from the upper half. Note that from Group 1 to Group 4, the F1-score distribution becomes more divergent and the median decreases. The methods to improve the predictions for Group 4 are a subject for further research.

To measure the difference in the prediction quality of applications from Groups 3 and 4, we used the PR AUC metric. This metric is commonly used to compare models for unbalanced problems. A precision-recall (PR) curve is a parametric curve $(x(t), y(t))$ where t is the threshold value and x, y are the values of recall and precision, respectively. The area under the curve (AUC) is an invariant used to compare model quality (the larger the AUC, the higher the prediction quality).

We used SPMLGen to generate a series of predictions for APK files from Groups 3 and 4, with threshold values ranging from 0 to 1 in steps of 0.001. For each value, the recall and precision were computed, providing a single point with coordinates (recall, precision) on the PR curve. The model based on Device profiles demonstrates good results when predicting applications from Group 3, but the quality of predictions for applications from Group 4 may vary greatly depending on the application. PR curves for applications from Group 3 (‘‘One Connect’’, ‘‘Write on PDF’’) and from group 4 (‘‘Music’’, ‘‘Calculator’’) are depicted in Fig. 8.

To better understand the prediction quality for applications that were not used in model training, we created three groups with 10 applications each and labeled them as groups A, B, and C. For each group, we trained a model that was unfamiliar with applications from two other groups. Fig.9 shows the results of cross-inference, where we applied each model to predict the hot methods of two groups that are unfamiliar to this model. The PR curves for the predictions of the applications in Group A using Model C are shown in Fig. 10.

B. APPLICATIONS SPEED-UP

To test the quality of the model generated using our method, we measured the launch time of the applications in each group. App launch time is the time required to load the main activity. We performed 20 launches of each application compiled with a quicken filter (during this type of compilation, the DEX instructions are optimized for better performance), with the SPMLGen profile (predicted with the device-specific model). Then, we took the median of each bunch of launches to neutralize the side effects (app execution on different CPU cores and frequencies and other processes). Let T_0, T_{spmlgen}, and T_{device} be the median launch time with a quicken filter, SPMLGen profile and Device profile, respectively. We define the relative speed-up R_{spmlgen}, R_{device} for SPMLGen and

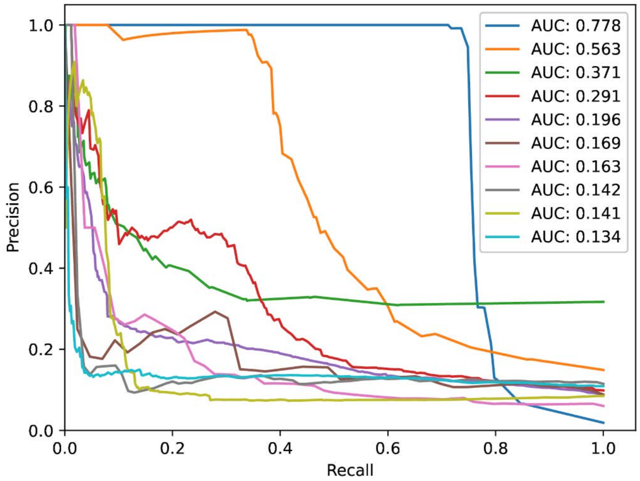

FIGURE 10. PR curves for predictions of group A using model C.

Cloud profiles as

$$R_{\text{spmlgen}} = \left(1 - \frac{T_{\text{spmlgen}}}{T_0}\right) \cdot 100\%,$$
$$R_{\text{device}} = \left(1 - \frac{T_{\text{device}}}{T_0}\right) \cdot 100\%.$$

The distribution of R_{spmlgen} and R_{device} are shown in Fig. 11. As we can see, the median values of speed-up of the SPMLGen and Device profiles for Groups 1–3 are approximately the same. However, the SPMLGen speed-up for Group 4 is less than that of the Device profiles. This is because our model predicts profiles for applications similar to those that it was trained on better than for completely different apps. Experiments show that the average speed-up for A52 is greater than that of S22. This is because S22 is a more powerful device that can afford to execute apps and perform JIT-compilation at the same time. Thus, our method can achieve more benefit on low-end devices, although it also provides considerable speed-up on flagships.

VII. LIMITATIONS

We encountered several difficulties in implementing our method. First, we could not use profiles collected during real customer usage because it requires the development of a special infrastructure for profile retrieval. Therefore, we used the Device profiles. We can see that SPMLGen can be successfully used to predict profiles. Second, it is challenging to construct an application execution speed metrics other than its start time.

Although SPMLGen provides considerable application launch speed-up, it has several drawbacks.

1) Our ML model attained a higher prediction quality for packages whose older versions were used in the training dataset. As for the applications not included in the dataset, the profile quality is lower, so is the speed-up.

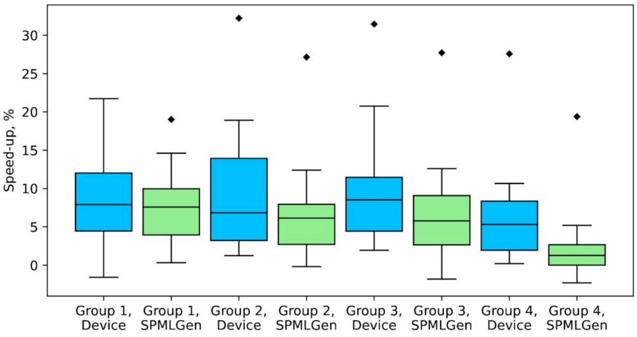

FIGURE 11. Relative speed-up distribution.

2) Inference time on device. The average application has thousands of samples, therefore we need prediction rates sufficiently high to run on devices within an appropriate time. To date, our method produces inference at 0.02 ms per sample, which can add up to 16 s for a rather large application of 850,000 methods. We plan to use GPU acceleration to speed up the inference; however, this work is challenging because of the lack of libraries working with random forests on a GPU.
3) Number of false positive results. Although the profile quality is good, it contains a relatively high number of false-positive classes and methods. Moreover, false positives accumulate in the profile when it is merged with the new one generated as a result of user behavior.
4) Because the profiles we use for training are obtained without user interaction with the app, we generate a profile that does not consider user behavior. We plan to use custom ML-based scripts to simulate human interactions when generating the training profiles.
5) Because the number of classes and methods that end up in a profile amounts to 5–20%, the datasets used for training are unbalanced. This could be the reason for the higher number of false positives. Furthermore, data augmentation for our type of datasets cannot be performed using existing algorithms.
6) APK developers may use code obfuscation techniques that may have an impact on the prediction quality.

VIII. CONCLUSION

Because the generation of the application profile in our method is based on ML rather than, for example, human-generated heuristics and rules, the profile generated by the ML-based model according to the present invention is more complete and potentially more universal. SPMLGen is applicable at any time when application code is available. Thus, it avoids delays caused by the need to collect information about the execution of the application. In addition, user privacy can be ensured with the proposed method because the prediction of performance characteristics can be performed without transferring any information outside the user device.

The proposed method is used by Samsung to generate profiles for preloaded applications during firmware building.

Thus, along with firmware updates, users receive optimized applications from Samsung. Our plans for the future include creating an ML model for smartphones and generating profiles on the device when the user updates an application or installs a new application. On-device ML is a modern technological trend that has both advantages (low latency, privacy, working in offline mode, no maintenance cost) and disadvantages (less powerful models due to size restrictions, restricted computing resources, and power consumption limitations). Despite these shortcomings, using SPMLGen directly on devices can complement or replace Google Cloud and improve the performance of PGO on Android in the future. Therefore, this is of great interest for further development.

ACKNOWLEDGMENT

The authors thank Aleksandra Soroka and Ivan Maidanskii for participating in the development of the SPMLGen method and the preparation of the patent [36].

REFERENCES

[1] Y. Wu, "Method and system for collaborative profiling for continuous detection of profile phase transitions," U.S. Patent 20 040 015 930 A1, Mar. 26. 2001. [Online]. Available: https://patents.google.com/patent/US20040015930A1/en

[2] T. M. Conte, B. A. Patel, K. N. Menezes, and J. S. Cox, "Hardware-based profiling: An effective technique for profile-driven optimization," *Int. J. Parallel Program.*, vol. 24, no. 2, pp. 187–206, Apr. 1996.

[3] B. B. Yilmaz, E. M. Ugurlu, F. Werner, M. Prvulovic, and A. Zajic, "Program profiling based on Markov models and EM emanations," in *Proc. SPIE*, Apr. 2020, pp. 69–83.

[4] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. 4th ACM SIGACT-SIGPLAN Symp. Princ. Program. Lang. (POPL)*, Los Angeles, CA, USA, 1977, pp. 238–252.

[5] T. Ball and J. R. Larus, "Branch prediction for free," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, New York, NY, USA, 1993, pp. 300–313.

[6] Y. Wu and J. R. Larus, "Static branch frequency and program profile analysis," in *Proc. 27th Annu. IEEE/ACM Int. Symp. Microarchit.*, San Jose, CA, USA, 1994, pp. 1–11.

[7] B. L. Deitrich, B. Chung Chen, and W. W. Hwu, "Improving static branch prediction in a compiler," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, Paris, France, 1998, pp. 214–221.

[8] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison, "Accurate static estimators for program optimization," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, New York, NY, USA, 1994, pp. 85–96.

[9] J. R. C. Patterson, "Accurate static branch prediction by value range propagation," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, New York, NY, USA, 1995, pp. 67–78.

[10] C. Boogerd and L. Moonen, "On the use of data flow analysis in static profiling," in *Proc. 8th IEEE Int. Work. Conf. Source Code Anal. Manipulation*, Beijing, China, 2008, pp. 79–88.

[11] R. P. L. Buse and W. Weimer, "The road not taken: Estimating path execution frequency statically," in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, Vancouver, BC, Canada, May 2009, pp. 144–154.

[12] Z. Wang, G. Tournavitis, B. Franke, and M. F. P. O'boyle, "Integrating profile-driven parallelism detection and machine-learning-based mapping," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 1, pp. 1–26, Feb. 2014.

[13] S. Zekany, D. Rings, N. Harada, M. A. Laurenzano, L. Tang, and J. Mars, "CrystalBall: Statically analyzing runtime behavior via deep sequence learning," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12.

[14] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin, "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks," in *Proc. ICML*, Long Beach, CA, USA, 2019, pp. 4505–4515.

[15] S. Sharma, A. Asthana, T. J. Mahaffey, and T. H. Tzen, "Profile guided optimization in the presence of stale profile data," U. S. Patent 20 160 004 518 A1 Jul. 3, 2014. [Online]. Available: https://patents.google.com/patent/US20160004518A1/en

[16] *Implementing ART Just-In-Time (JIT) Compiler*. Accessed: Aug. 15, 2022. [Online]. Available: https://source.android.com/devices/tech/dalvik/jit-compiler

[17] K. Semenova, R. Ravikumar, and C. Craik. I*Mproving App Performance With Baseline Profiles*. Accessed: Aug. 15, 2022. [Online]. Available: https://android-developers.googleblog.com/2022/01/improving-app-performance-with-baseline.html

[18] C. Juravle. *Improving App Performance With ART Optimizing Profiles in the Cloud*. Accessed: Aug. 15, 2022. [Online]. Available: https://android-developers.googleblog.com/2019/04/improving-app-performance-with-art.html

[19] K. Jeong, S. Lonchakov, I. Titarenko, G. Arakelov, I. Maidanskii, H. Kim, and A. Semuka, "Method and apparatus for improving runtime performance after application update in electronic device," Accessed: Sep. 6, 2019. U.S. Patent WO2 021 045 428 A1. [Online]. Available: https://patents.google.com/patent/WO2021045428A1

[20] K. Jeong, S. Lonchakov, I. Titarenko, I. Maidanskii, and K. Jeon, "Application installation method and electronic device for supporting same," U.S. Patent WO2 022 030 903 A1, Aug. 3, 2020. [Online]. Available: https://patents.google.com/patent/WO2022030903A1

[21] Y. Bengio, Y. LeCun, and G. Hinton, "Deep Learning," *Nature*, vol. 521, pp. 436–444, May 2015.

[22] L. Rokach, "Ensemble-based classifiers," *Artif. Intell. Rev.*, vol. 33, nos. 1–2, pp. 1–39, Feb. 2010.

[23] M. N. Adnan and M. Z. Islam, "Forest PA: Constructing a decision forest by penalizing attributes used in previous trees," *Exp. Syst. Appl.*, vol. 89, pp. 389–403, Dec. 2017.

[24] J. Jia, Z. Liu, X. Xiao, B. Liu, and K.-C. Chou, "iPPI-Esml: An ensemble classifier for identifying the interactions of proteins by incorporating their physicochemical properties and wavelet transforms into PseAAC," *J. Theor. Biol.*, vol. 377, pp. 47–56, Jul. 2015.

[25] A. A. Taha and A. Hanbury, "Metrics for evaluating 3D medical image segmentation: Analysis, selection, and tool," *BMC Med. Imag.*, vol. 15, no. 1, pp. 1–28, Aug. 2015.

[26] G. W. Brier, "Verification of forecasts expressed in terms of probability," *Monthly Weather Rev.*, vol. 78, no. 1, pp. 1–3, Jan. 1950.

[27] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognit. Lett.*, vol. 27, no. 8, pp. 861–874, Jun. 2006.

[28] J. Davis and M. Goadrich, "The relationship between precision-recall and ROC curves," in *Proc. 23rd Int. Conf. Mach. Learn. (ICML)*, New York, NY, USA, 2006, pp. 233–240.

[29] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[30] D. Yates and M. Z. Islam, "FastForest: Increasing random forest processing speed while maintaining accuracy," *Inf. Sci.*, vol. 557, pp. 130–152, May 2021.

[31] L. Breiman, "Bagging predictors," *Mach. Learn.*, vol. 24, no. 2, pp. 123–140, Aug. 1996.

[32] J. A. Aslam, R. A. Popa, and R. L. Rivest, "On estimating the size and confidence of a statistical audit," in *Proc. EVT* Boston, MA, USA, 2007, p. 8.

[33] E. Bauer and R. Kohavi, "An empirical comparison of voting classification algorithms: Bagging, boosting, and variants," *Mach. Learn.*, vol. 36, pp. 105–139, Jul. 1999.

[34] D. Larose and C. Larose, "Decision trees," in *Discovering Knowledge in Data: An Introduction to Data Mining*, 2nd ed. New York, NY, USA: Wiley, 2014, ch. 8, pp. 174–179.

[35] K. Boyd, K. H. Eng, and C. D. Page, "Area under the precision-recall curve: Point estimates and confidence intervals," in *Proc. Joint Eur. Conf. Mach. Learn. Knowl. Discovery Databases*, Prague, Czech Republic, 2013, pp. 451–466.

[36] A. D. Soroka, S. A. Pavlova, and I. S. Maidanskii, "Method for generating a program profile based on machine learning for profile optimization of the program and an electronic device implementing IT," U.S. Patent 2 778 078 C1, Aug. 15, 2022.

ANDREI VISOCHAN received the B.S. degree in mathematics and applied mathematics from Novosibirsk State University, Novosibirsk, in 1994.

From 2014 to 2018, he was a Lead Software Engineer in cartography services, with 2GIS. Since 2018, he has been a Project Manager with the Platform Laboratory, Samsung Research Russia, Moscow. His main research interests include Android platform, Android runtime, and ML on code.

ANDREY STROGANOV received the Specialist degree in microelectronics and solid-state electronics from Russian Technological University, Moscow, in 2008, and the Ph.D. degree in mathematics from the Institute of Artificial Intelligence, Russian Technological University, in 2013.

From 2013 to 2022, he was an Assistant Professor with the Institute of Artificial Intelligence, Russian Technological University. Since 2022, he has been a Leading Software Engineer with the Platform Laboratory, Samsung Research Russia, Moscow. He is the author of more than 20 articles in differential equations, mathematical modeling and parallel computations. He used to be interested in combinatorics, mathematical modeling, and discrete mathematics. His current research interests include software optimization, ML, and data science.

IVAN TITARENKO received the B.S. degree in information security from the Moscow Power Engineering Institute, Moscow, in 2020.

Since 2018, he has been a Software Engineer with the Platform Laboratory, Samsung Research Russia, Moscow. He is the author of two inventions (WO2021045428A1 and WO2022030903A1) regarding Android application performance improvements. His research interests include system software development, Java runtime, application performance issues, Android platform, and garbage collection.

Mr. Titarenko was a recipient of Samsung Mobile Research and Development CTO Award, in 2020.

SERGEI LONCHAKOV received the Specialist and M.S. degrees in physics from Lomonosov Moscow State University, Moscow, in 2016.

Since 2018, he has been a Software Engineer with the Platform Laboratory, Samsung Research Russia, Moscow. He is the author of two inventions (WO2021045428A1 and WO2022030903A1) regarding Android application performance improvements. His main research interests include system software development, Java runtime, and Android platform.

Mr. Lonchakov was a recipient of Samsung Mobile Research and Development CTO Award, in 2020.

STANISLAV MOLOGIN received the B.S. and M.S. degrees in applied mathematics and computer science from Lomonosov Moscow State University, Moscow, in 2021.

Since 2018, he has been a Junior Software Engineer with the Platform Laboratory, Samsung Research Russia, Moscow. His main research interests include system software development and Android platform.

Mr. Mologin was a recipient of Samsung Mobile Research and Development CTO Award, in 2020.

SVETLANA PAVLOVA received the B.S. degree in fundamental and applied linguistics from the Higher School of Economics, Moscow, in 2017.

Since 2018, she has been an Engineer with Samsung Research Russia, Moscow. She is the author of ‘‘Cross-Lingual Named Entity List Search via Transliteration’’ (Proceedings of the 12th Language Resources and Evaluation Conference, 2020). Her main research interests include natural language processing, voice assistants, chatbots, search engines, and MLOps.

ANASTASIA LYUPA received the B.S. and M.S. degrees in applied mathematics and physics from the Moscow Institute of Physics and Technology, Moscow, in 2014.

From 2013 to 2022, she was a Software Engineer with Samsung Research Russia. Since 2022, she has been a Senior Engineer with the Moscow Software OS Laboratory, Huawei Technologies Russia, Moscow. She is the author of more than seven papers in various journals and at international conferences on the simulation of multiphase flows on high-performance computing systems. She used to be interested in mathematical modeling and parallel computations. Her main research interests include system software development and software optimization.

ANNA KOZLOVA received the M.S. degree in operations research and system analysis from St. Petersburg University, St. Petersburg, in 2021.

Since 2021, she has been an Assistant Engineer in machine learning with Kelly Services CIS, Moscow. Her main research interests include explainable artificial intelligence, multi-agent reinforcement learning, and on-device machine learning.

• • •