MiniMon: Minimizing Android Applications with Intelligent Monitoring-Based Debloating

Jiakun Liu
Singapore Management University
Singapore
jkliu@smu.edu.sg

Zicheng Zhang
Singapore Management University
Singapore
zczhang.2020@phdcs.smu.edu.sg

Xing Hu
Zhejiang University
Hangzhou, Zhejiang, China
xinghu@zju.edu.cn

Ferdian Thung*
Singapore Management University
Singapore
ferdianthung@smu.edu.sg

Shahar Maoz
Tel Aviv University
Israel
maoz@cs.tau.ac.il

Debin Gao
Singapore Management University
Singapore
dbgao@smu.edu.sg

Eran Toch
Tel Aviv University
Israel
erant@tauex.tau.ac.il

Zhipeng Zhao
Singapore Management University
Singapore
zpzhao@smu.edu.sg

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

ABSTRACT

The size of Android applications is getting larger to fulfill the requirements of various users. However, not all the features of the applications are needed and desired by a specific user. The unnecessary and non-desired features can increase the attack surface and consume system resources such as storage and memory. To address this issue, we propose a framework, MiniMon, to debloat unnecessary features from an Android app based on the logs of specific users' interactions with the app.

However, rarely used features may not be recorded during the data collection, and users' preferences may change slightly over time. To address these challenges, we embed several solutions in our framework that can uncover user-desired features by learning and generalizing from the logs of how users interact with an application. MiniMon first collects the application methods that are executed when users interact with it. Then, given the collected executed methods and the call graph of the application, MiniMon applies 10 techniques to generalize from logs. These include three program analysis-based techniques, two graph clustering-based techniques, and five graph embedding-based techniques to identify the additional methods in an app that are similar to the logged executed methods. Finally, MiniMon generates a debloated application by removing methods that are not similar to the executed methods. To evaluate the performance of variants of MiniMon that use different generalization techniques, we create a benchmark for a controlled experiment. The results show that the graph embedding-based generalization technique that considers the information of all nodes in the call graph is the best, and can correctly uncover 75.5% of the unobserved but desired behaviors and still debloat more than half of the app. We also conducted a user study that uncovers that the use of the intelligent (generalization) method of MiniMon boosts the overall user satisfaction rate by 37.6%.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Android, Software Debloating, Log Analysis

ACM Reference Format:
Jiakun Liu, Zicheng Zhang, Xing Hu, Ferdian Thung*, Shahar Maoz, Debin Gao, Eran Toch, Zhipeng Zhao, and David Lo. 2024. MiniMon: Minimizing Android Applications with Intelligent Monitoring-Based Debloating. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639113

1 INTRODUCTION

Android reigns in the mobile operating system market worldwide nowadays, with a 71.95% market share in Q1 2023 [4]. With the increasing performance of CPU and the storage of mobile devices, Android applications (i.e., apps) are getting larger because developers would like to add more features (i.e., a feature is -functionality that satisfies a certain requirement [59]) to fulfill the requirements of various users. For example, the size of the Twitter (currently known as X) app is 108 MB. Users can use Twitter to tweet, retweet, reply, share, explore top trending topics, and even converse with sound [6, 13].

However, not all the features of the apps are needed by users, resulting in a bloated app. Prior studies showed that 80% of features in average software products are rarely or never used [15]. These unused features can increase the attack surface and cost additional resources, such as storage or memory [22]. More specifically, while certain features may be essential for some users, they may not be as important for others. For example, Figure 1 shows that a user

* Corresponding author.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639113

Figure 1: An example of a user complaining about a Twitter's feature. This example shows that different users have different requirements for the "bookmark" feature.

complained about a Twitter's feature. The bookmark feature is helpful to some users but not useful to others. This indicates that different users have different preferences.

To address the contradiction between developers' desire to attract more users with more features and users' preference for using only a subset of those features, prior studies proposed several solutions to debloat Android apps, by removing a specific feature (e.g., at the granularity of Activity, Permission, and Modularity) [59], or removing dead code statically [44] or dynamically [51]. Huang et al. proposed a UI-based approach to remove user-specified unused UI components and the corresponding backend code [43]. They asked users to manually label the unwanted components. However, identifying all unwanted components from all components in the app is difficult because (1) components can be created dynamically [14], and (2) it is difficult to fully explore the app to obtain all components. The cutting-edge approach does not have a high coverage at the activity level (i.e., 68%) [47], not to mention at the components level. Therefore, there can be unwanted components never explored by users. Users are not aware of the existence of these features, let alone actively labeling them as unwanted components. Therefore, this motivates us to ask users to actively label the desired features, rather than labeling the unwanted components. Besides, Huang et al.'s work requires developers to manually obtain the IDs of the specified UI elements to map the component to the backend code. This reduces the automation level of the tool. This motivates us to automatically map the UI elements to the backend code.

To fill the gap, we propose MiniMon, which can preserve the user-desired features and remove the ones not used by end users. MiniMon first ❶ instruments apps statically. When users explore the instrumented app, the executed app **methods** are recorded in the log. By doing so, we can (1) collect the user-desired features and (2) the corresponding backend methods of the user-desired features during the interaction. However, some user behaviors may not be recorded during monitoring (e.g., the ones related to rarely used features), and users' preferences can be slightly changed [66]. For example, Figure 1 shows that the user may not use the "bookmark" feature during monitoring, but he/she may want to use the "bookmark" feature in the future. This motivates us to generate a debloated app that includes user-desired features that are exercised by unseen user behaviors.

To identify the methods of user-desired features that are exercised by unseen user behaviors, MiniMon ❷ generates the call graph (shortened as CG) of the apps. After this step, one way to proceed is to employ semi-supervised learning to predict which methods are related to the user-desired feature, using graph node classification approaches [39, 45, 46, 50, 56, 60, 67, 70]. However, we cannot apply these solutions because in the semi-supervised graph nodes classification task, there is more than one label in the labeled data, i.e., there are negative samples. Unfortunately, for our task, we only have the methods that are executed when users explore the app, and we do not have the methods that will never be executed. Thus, ❸ given the CG and the methods of the features that are recorded during monitoring, MiniMon needs to identify additional methods of desired features using an unsupervised technique. After that, MiniMon ❹ can remove the methods that are not related to the execution of the desired features and generate a debloated app.

We refer to the component in MiniMon that identifies the methods of desired features as **MethodGeneralizer**. We need data to build the model for MethodGeneralizer. However, there is no such dataset that records the executed methods corresponding to a set of similar user behaviors. To solve this problem, we create a benchmark recording the interaction of 214 similar user behaviors and the corresponding executed methods in 45 Android apps with the help of SARA [40]. We then evaluate the performance of MethodGeneralizer in a controlled experiment by considering a user behavior as an unseen one and the others as the recorded ones. Given the executed methods recorded during the monitoring of the 45 apps and their corresponding CGs, we investigate three research questions:

- **RQ1: How effective are different variants of MethodGeneralizer in identifying methods of desired features?**
We apply three program analysis-based techniques, two graph clustering-based techniques, and five graph embedding-based techniques as the technique in MethodGeneralizer components. We empirically evaluate each MethodGeneralizer technique on our benchmark in terms of Recall (ability to uncover methods of desired features) and Debloating Rate (proportion of methods that are removed), and the weighted harmonic mean of Recall and Debloating Rate (i.e., $F_{debloat}$). The evaluation results show that the four graph embedding-based techniques (Node2Vec and inverse-document-frequency techniques, except LSTM) outperform program analysis-based techniques and graph clustering-based techniques in terms of all metrics. MiniMon can debloat the apps by 58% on average and can uncover 76% of methods of desired features using the best-performing MethodGeneralizer technique.

- **RQ2: Can MiniMon effectively debloat apps?**
We replay the interactions recorded in the benchmark, on the debloated apps that are generated by the best-performing MethodGeneralizer. We also conduct a user study using 8 apps and collect how users use the apps from 8 users. We find that 88.8% of the test cases succeed to replay. The user study results demonstrate that the

use of the intelligent (generalization) method of MiniMon boosts the overall user satisfaction rate by 37.6%.

We also discuss the most important component for the best-performing MethodGeneralizer techniques (graph embedding-based techniques) using an ablation study. We find that considering all nodes in the call graph is the most important component in graph embedding-based techniques.

In summary, the contributions of this paper include:

- We build a benchmark with 214 user behaviors and the corresponding executed methods of 45 Android apps.
- We implement a monitoring-based-debloating framework, MiniMon, to generate a debloated app that can cover as many user-desired features as possible.
- We propose MethodGeneralizer, a component of MiniMon to identify methods of desired features that are exercised by unseen user behaviors, given the CG and the recorded executed methods during monitoring. We apply 10 techniques in MethodGeneralizer and empirically evaluate their performance. Results show that the graph embedding-based technique that considers the global information of the call graph can effectively identify additional methods of the desired features.

The remainder of this paper is organized as follows: Section 2 describes the monitoring-based debloating framework MiniMon and its usage scenarios. We elaborate on MethodGeneralizer which identifies methods in an app corresponding to user-desired features but are not logged during monitoring in Section 3. We describe how we build our benchmark in Section 4. Section 5 presents the evaluation settings. In Section 6, we present the evaluation results. We discuss the most important component in the best-performing MethodGeneralizer and the threats to validity in Section 7. After a review of related work in Section 8, we conclude this paper and point out future work in Section 9.

2 MINIMON: MONITORING-BASED DEBLOATING FRAMEWORK

Figure 2 presents an overview of MiniMon, our monitor-based debloating framework. MiniMon first instruments (i.e., insert code) the Android app to monitor its execution, and users can explore the instrumented app (Section 2.1). Meanwhile, MiniMon uses static analysis to get the call graph of the app (Section 2.2). Based on the call graph and the recorded executed methods, the MethodGeneralizer component identifies additional methods that the user is also likely to use for the desired features (Section 3). Finally, MiniMon prunes the code instructions that are not necessary for the execution of the desired features (Section 2.3).

2.1 Instrumenting Android Applications

To understand how users interact with the app, we collect the exploration trace of the users by collecting the methods that are executed in the exploration process. We do not collect it visually (e.g., screen recording) because it may collect sensitive information from users and may require more storage. Instead, we focus on the methods that are executed during monitoring. This is because if we focus on the class level, we cannot perform fine-grained app debloating: many functions may only be related to a single method.

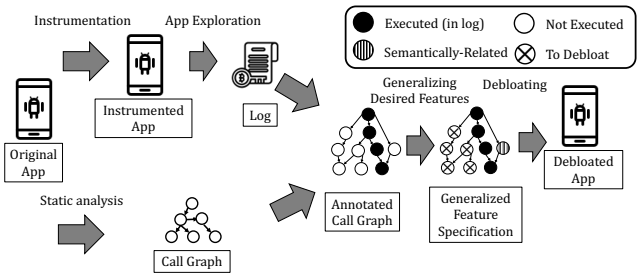


Figure 2: Overview of monitor-based debloating

If we focus on the instruction level, we would need to record a lot of information, which would require a lot of storage. We believe that the method granularity is a suitable choice.

To do so, we statically instrument the Android apps. We cannot use the real user monitoring tools that instrument on the source code (e.g., Amplitude [18], and Dynatrace [1]) because we do not have the source code of the app. Besides, we do not instrument the apps dynamically because the dynamic instrumentation requires a lot of technical work, i.e., (1) root access to the Android device, and (2) connection to the computer when exploring the app [6]. However, we cannot request users to do such technical work as they may know nothing about computer science. So, we use the BodyTransformer class in Soot to statically instrument the app by inserting the (1) method signature, (2) the time stamp, and (3) the app name, before the first instruction of each method [7]. Once the method is executed, the inserted code will be executed and the method signature, the time stamp, and the app name will be recorded in the log. In our controlled experiment, we use Logcat to collect the log from the Android emulator. In practice, we ask the user to install the Logcat record tools from Google Play, e.g., MatLog[1]. Similar to popular real-user monitoring tools (e.g., Amplitude [18], and Dynatrace [1]), the log would be buffered in the RAM of the Android device and would be batch-written to the disk, to minimize the impact on the performance of the app.

2.2 Generating Call Graph

We use FlowDroid to perform the static analysis to generate the call graph (i.e., CG) [19]. The nodes in the CG represent methods, and the edges represent the relationship between methods (caller→callee). Following prior studies, we use (1) lifecycle methods (e.g., onCreate(), onStop()) in Android components (e.g., Activity), and (2) callbacks (e.g., location updates or UI interactions), as the entry points of an app [19, 59]. In Android, lifecycle methods are the standard entry points to components. The execution of callbacks in Android (updating the location or clicking the button in an Activity) can trigger the execution of app code. After that, we traverse the explicit method calls to build the CG [54].

However, Android apps also allow implicit method calls. For reflection, to identify the target of the reflection, we identify all propagated string constants [19]. For Inter-component Communication (i.e., ICC), we use ICCBot to detect ICC in an app [68]. Compared with traditional ICC resolution tools (e.g., Epicc [49]

[1]https://play.google.com/store/apps/details?id=com.pluscubed.matlog

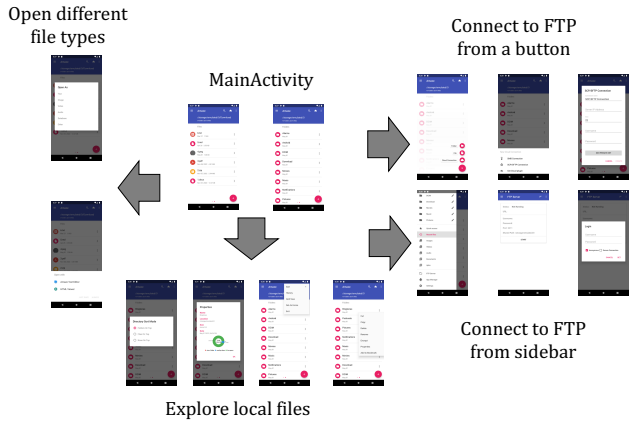


Figure 3: Open different file types, browse local files, and connect to FTP features of Amaze File Manager app.

and IC3 [48]), ICCBot is a fragment-aware and context-sensitive ICC resolution tool. For asynchronous task, we follow Tang et al. [59] and add the following edges to CG: AsyncTask.execute() → onPreExecute, and onPreExecute → doInBackground, etc.

2.3 Debloating Android Application

After identifying the to-keep methods and the to-remove methods (in Section 3), we remove the to-remove methods from the Android app using Soot. Following a prior study that debloat Java program [27], for each to-remove method, we provide two options: (1) clear the body of the method and change the return value to null (when the return type is a class of the reference type of Java) or to 0 (when the return type is a numeric class in Java), or (2) replace the method body with code that notifies that the explored functionality has been removed. Following a prior study [27], we use the first option by default, and our results in Section 6 are for this option. Still, a user may choose the other option. We did not change the UI elements of the app so that users can see the original features of the app. Note that the graph embedding-based method in Section 3 measures the similarity between all app methods and the methods executed during monitoring. We set a series of similarity thresholds. For each threshold, we identify a set of similar methods. This indicates that we have multiple sets of methods, with each set corresponding to a similarity threshold. Therefore, for the embedding-based method in Section 3, we will have multiple versions of the Android app with different similarity thresholds.

2.4 Usage Scenario

Figure 3 shows the possible interaction in an example app (Amaze File Manager) [3]. Consider a user Bob, who just wants to browse the local files on Amaze File Manager. Without our tools, Bob will install the full Amaze File Manager APK. The user's unwanted features, e.g., connecting to the FTP, will be loaded into the memory. The execution of these additional features will consume the battery and the storage of Bob's phone. If Bob uses our tools, Bob can install the debloated APK, which only contains the features that Bob wants. More specifically, Bob will need to install the instrumented APK first, and then explore the app for daily use. Then, Bob sends the log to our server, and our server will send back debloated Amaze File Manager APKs that are debloated using different similarity thresholds to Bob. The APK that Bob chooses to install will depend on his risk profile and willingness to do trial and error. If Bob cares a lot about security and app size, he can pick the APK that is the most debloated. This may imply that he may need to pick another APK if the debloating is overly restrictive. Otherwise, if Bob does not want to do much trial-and-error, he can pick an APK that is debloated conservatively, allowing Bob to perform behaviors that are marginally related to the features.

3 METHODGENERALIZER: GENERALIZING DESIRED METHODS

3.1 Our Task

Given a set of methods M which are all the methods in an app, a set of the executed methods m that are collected when users explore the app, our goal is to identify a set of methods M' which are the methods of all user-desired features (our goal is not to label the features from the methods recorded in traces [17, 20, 37, 65]). By doing so, we can include the rare use but user-desired features that are exercised by unseen user behaviors in the debloated app (i.e., generalization). More specifically, we have $m \subseteq M' \subseteq M$.

3.2 Our Techniques for MethodGeneralizer

Prior studies on program comprehension [16, 23, 41, 55, 58, 61] have shown the effectiveness of using method-call relationships to infer feature-relatedness (i.e., whether two methods are related to the same feature). For example, Biggerstaff et al. [23] and Alanazi et al. [16] used method-call relationships to identify the mappings between source code and features. Motivated by these studies, we use method-call relationships to infer feature-relatedness. Specifically, based on method-call relationships captured in a call graph and a set of executed methods, MethodGeneralizer infer additional feature-related methods. We consider three program analysis-based techniques (i.e., forward-slicing, backward-slicing, and forward-backward-slicing) [62], two graph clustering-based techniques (i.e., Louvain community detection [25], and Label propagation algorithm [71]), and five graph embedding-based techniques (i.e., node2vec [39], LSTM [42], OneHot Encoding, IDF-Encoding, and IDF-POS encoding) as the candidate technique for MethodGeneralizer.

3.2.1 *Program analysis-based techniques.*

- **Forward slicing:** (shortened as FORWARD) Intuitively, if a method m is currently executed, then the methods that are called by m also can be executed in the future. For example, if two methods are called by the same set of methods (e.g., invoked by the same entry point), they are likely to be similar (e.g., open different types of files in Amaze file manager). More specifically, we add the methods that are (in)directly called by the methods in m to M'.
- **Backward slicing:** (shortened as BACKWARD) Intuitively, if a method m is currently executed, then the methods that call m also can be executed in the future (multiple entry points). Similarly, if two methods call the same set of methods then they are likely to be similar (e.g., use the FTP feature from FTP Activity, and FTP

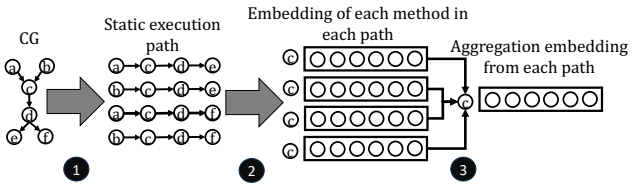


Figure 4: The flow to generate the method embedding by converting CG to sequences

connection button). More specifically, we add the methods that are (in)directly call the methods in m to M'.

• **Forward and backward slicing:** (shortened as BIDIRECTION) The intuition is that if a method m is currently executed, then the methods that are called by m and the methods that call m also can be executed in the future. More specifically, we add the methods that are reachable by the methods in m to M'.

3.2.2 Graph clustering-based techniques.

• **Louvain community detection algorithm:** (shortened as LCD) According to the principle of object-oriented programming, methods with high cohesion and low coupling (i.e., high modularity) can be considered a module. We consider a module in a program to be a feature for users and employ LCD Algorithm to identify modules. LCD Algorithm first assigns every node to be in its community. Then, it iteratively determines whether moving a node to its neighbor communities can have better modularity. If so, the node would be moved to its neighbor community.

• **Label propagation algorithm:** (shortened as LPA) The intuition is that the method execution process is similar to the label propagation, i.e., if its neighbors are executed, then the method is likely to be executed. Therefore, we use LPA to identify similar methods. More specifically, LPA initializes each node with a unique label, then repeatedly sets the label of a node to be the label that appears most frequently among the neighbors.

We follow the settings in Tang et al.'s work and set the weight of an edge to be 0.5 or 1. 0.5 indicates that the invocation is unidirectional (i.e., either m invokes n or n invokes m). 1 indicates that the invocation is bidirectional (m invokes n and n invokes m). We apply LCD and LPA to CG, with other parameters set to default[9, 10]. If there is an executed method in the community generated by LCD or LPA, we add all the methods in the community to M'.

3.2.3 Graph embedding-based techniques. Here, we would like to ❶ generate the embedding for each method in CG and then ❷ use these embeddings to identify the methods of user-desired features that are exercised by unseen user behaviors. Intuitively, if we can map the multitude of information in CG (e.g., structure information, calling relationships) to a low-dimensional vector space, we can leverage such information to better identify the methods of desired features. We first explore how to generate the method embedding by converting CG to sequences. Figure 4 shows the flow to generate the method embedding by converting CG to sequences. Specifically, to generate the embedding for each method in CG, we (1) first convert the graph structure to sequences and (2) apply 4 techniques (i.e., LSTM, One-hot, IDF, and IDF-POS) to generate the embedding for each method in each sequence. Then, we (3) aggregate the embeddings for each method across sequences to obtain the final embedding of each method. We also use a generic graph embedding technique, Node2Vec [39], to directly generate the embedding of the nodes in the CG. Finally, given the embedding of methods in CG, we use the cosine similarity to identify the additional methods of user-desired features.

Generating Static Execution Path: We use the CG to obtain the static execution path of each app. **Static Execution Path** is a **sequence** of methods that are connected by the edges in a CG. The first method in the static execution path is the entry point of the app (the in-degree of each entry point method is 0). Then, we perform a depth-first search (DFS) on the CG to obtain the static execution path. To deal with recursion, each edge will only be added once. Figure 4 shows an example CG and its static execution paths. By doing so, we convert the graph structure of the CG into a sequence structure and keep the CG topology. For example, the nodes in CG with a higher degree of centrality, and the nodes that connect to these nodes, are in more static execution paths. The static execution path records all possible execution sequences.

Generating the Embedding of Each Method in Each Static Execution Path: Since the static execution path embodies the information of CG, we would like to obtain embeddings for the methods in the CG using embeddings from the static execution path. More specifically, we explore the following techniques:

• **LSTM** is a recurrent neural network that can learn long-term dependencies in a sequence [42]. We consider the static execution path as a sequence and use LSTM model to learn the long-term dependencies between the methods in the execution path. We use the last hidden state of the LSTM model as the embedding of the execution path. Specifically, for each method in a static execution path, we divide this path into two parts, i.e., the path that (in)directly calls the method (i.e., caller path) and the path that is (in)directly called by the method (i.e., callee path). Then, we use the LSTM model to generate the embeddings of the caller path and the callee path, respectively. Finally, we concatenate these embeddings as the embedding of the method in this path.

• **One-hot** (shortened as Onehot) encoding takes all nodes in the path into consideration. For each method in the path, its embedding is considered with its context:

$$\text{Embedding}(m_i) = [\mathbb{1}(m_1), \ldots, \mathbb{1}(m_{i-1}), 0, \mathbb{1}(m_{i+1}), \ldots, \mathbb{1}(m_n)]$$

$\mathbb{1}(m_j)$ is the indicator function of the method m_j. If the method m_j is in the context of m_i, then $\mathbb{1}(m_j) = 1$; otherwise, $\mathbb{1}(m_j) = 0$.

• **IDF** The intuition is if a method is commonly executed in the static execution paths, then it should have a lower weight in the embedding. For example, a utility method, e.g., the amaze file app rewrites the ImmutableEntry class, is commonly executed in static execution paths. ImmutableEntry class can be frequently executed when the Map class is executed. When ImmutableEntry class is executed when using a certain feature, it does not indicate that the user would like to use another feature. In contrast, the $\langle com.amaze.filemanager.asynchronous.services.ftp.FtpService : voidrun() \rangle$ method is rarely executed in the static execution paths (52 out of 12,363 static execution traces). The execution of this method indicates the user is exploring the FTP connection

feature. This motivates us to involve the method frequency in generating embeddings of methods. Inspired by IDF (inverse document frequency) in information retrieval, we define the **inverse document frequency (IDF) of a method** as $\text{IDF}(m) = \log \frac{N}{n_m}$. The more static execution paths contain the method, the smaller the IDF (i.e., importance) of the method. The embedding of a method is

$$\text{Embedding}(m_i) = [\mathbb{1}(m_1) \times \text{IDF}(m_1), \cdots, \mathbb{1}(m_{i-1}) \times \text{IDF}(m_{i-1}), 0, \mathbb{1}(m_{i+1}) \times \text{IDF}(m_{i+1}), \cdots, \mathbb{1}(m_n) \times \text{IDF}(m_n)]$$

• **IDF-POS** IDF-POS considers the position information relative to the to-embedding nodes. If a method is executed, then the methods closer to the to-embedding method are more likely to be executed than the methods that are far away from the to-embedding method. Therefore, we define the **weight of a method m_q in the embedding of method m_p** as $\text{Weight}^{m_q}_{m_p} = q/p$ (if q<p) or $(n-q+1)/(n-p+1)$ (if q>p). Therefore, we have

$$\text{Embedding}(m_i) = [\mathbb{1}(m_1) \times \text{IDF}(m_1) \times \text{Weight}^{1}_{m_i}, \cdots, \mathbb{1}(m_{i-1}) \times \text{IDF}(m_{i-1}) \times \text{Weight}^{i-1}_{m_i}, 0, \mathbb{1}(m_{i+1}) \times \text{IDF}(m_{i+1}) \times \text{Weight}^{i+1}_{m_i}, \mathbb{1}(m_n) \times \text{IDF}(m_n) \times \text{Weight}^{n}_{m_i}]$$

Aggregating Embedding From Each Path: However, Figure 4 shows that a method can be in different static execution paths. This can lead to multiple embeddings of each method from different static execution paths. Therefore, we aggregate the embeddings from different static execution paths to obtain the embedding of a specific method. To do so, we use the max value of the embeddings of the methods in different static execution paths to obtain the embedding of a specific method (i.e., max pooling). By doing so, the embedding of a method is the most important method in the context of the method in the static execution paths.

• **Node2Vec** (shortened as N2V) Node2Vec is a graph embedding method that directly maps the nodes of a graph into a low-dimensional vector space [39] (i.e., without using the steps described in Figure 4). Different from the aforementioned techniques that generate sequences using DFS, Node2Vec is based on a combination of random walks to convert a graph to sequences. Then it uses a neural network, e.g., Word2Vec, to generate embedding based on sequence, which allows it to capture the properties of the graph from the nearby k windows' tokens. When the parameters of Node2Vec are set to large enough, it will generate sequences from the entry points to the end (same as ours), and the embedding of a method can be obtained from the information of the whole graph (same as ours, but with different embedding techniques: Node2Vec employs Word2Vec, while we use IDF information). We use the default settings of Node2Vec, and the input is the CG [31].

Finding Similar Methods: After obtaining the embeddings of each method, we need to find the methods of user-desired features M' given the executed methods m when monitoring. If we train a prediction model using the (semi-)supervised-based approach, then in the training dataset, the methods in m are labeled as 1, and there is no data labeled as 0 (as there is no method that must be not executed by the user). This motivates us to use the unsupervised approach to find the methods M' to the methods in m.

Moreover, different users have different perceptions of debloating. For example, some users may prefer to use different features to achieve the same goal (e.g., users can use the FTP feature from different entry points, such as FTP Activity, FTP connection button, and FTP Fragment), or the same actions to the app correspond to different features (e.g., open different types of files using amaze file manager). This indicates that such users have more user behaviors for the feature and we need to provide them with a larger M'. Another user may prefer to use the same feature as when we collect the data. This indicates that such users have fewer user behaviors for the feature and we need to provide them with a smaller M'.

In our work, we identify the methods that are similar to the executed methods m by calculating the Cosine similarity between the embedding of the executed methods m and the embedding of all the app methods. If we set different thresholds for Cosine similarity, we can obtain different sets of M' with different sizes. Thus, we can provide different users with the debloated app with different sizes of M'. Users can choose the debloated app that contains all desired features. More specifically, if the Cosine similarity between the embedding of a method and the embedding of the executed methods m is larger than a threshold (thresholds range from 0 to 1 with step 0.1), then we consider the method is similar to the executed methods m, and we add the method to the set M'. If there is a method in M' but none of its callers in M', we involve the caller with the highest similarity to M'. By doing so, we can identify the methods in M' with different similarity thresholds.

4 BENCHMARK CREATION

A user may only exercise a subset of behaviors that map to a desired feature. MiniMon then needs to generalize from this subset of behaviors to all the behaviors that belong to the desired feature. To test the ability of MiniMon to do this generalization in a more controlled setting involving a substantially large number of apps and features, we build a benchmark.

In this benchmark, for each app, we define a set of features. For each feature, we identify related user behaviors that map to the feature. For example, "Open File" is a feature considered in the leftmost part of Figure 3. For this feature, we have a set of user behaviors, e.g., "open a PDF file", "open a text file", "open a PPT file", etc. For each such user behavior (test case), we execute the behavior and collect a log of methods that are executed. To use the benchmark to test the generalization ability of MiniMon, given a target desired feature F and its corresponding set of user behavior logs (LF), we take a subset of logs in LF ($LF' \subset LF$) and input it to MiniMon. We then observe the ability of MiniMon to uncover methods that appear in $LF \setminus LF'$. These correspond to methods that are needed by the target desired feature but are not observed during the prior interactions that the user has with the app.

We collect 45 apps from Google Play. These apps appear among the top 40 apps in a category listed in AndroidRank.[2] The apps have an average of 15,318 lines of decompiled code and up to 67,312 lines of decompiled code. This average is larger than the average lines of code of applications in the benchmarks used to evaluate prior debloating studies [27, 53]. Note that these apps can be considered bloated if users only use some features; the unused features carry no benefit to the users and can be omitted for security [24, 38, 52, 57] and other reasons.

[2] https://www.androidrank.org/

For each app, we want to collect executed methods of a set of similar user behaviors. According to the Material Design guide [11], widgets in the same Activity are more likely to be related to each other [69]. Therefore, we consider the widgets in the same activity as a set of similar user behaviors.

Based on the above heuristics, the authors create test cases capturing related user behaviors exercising features of apps. These test cases are created by manually exploring the app. We use SARA [40] to record the test cases. SARA is a record-and-replay tool that (1) records motion events based on screen coordinates, and (2) replays recorded events based on both screen coordinates and widgets.

While exploring an app, the first or second author records the app launch (i.e., opening the app). By doing so, we ensure that the debloated apps can be opened. Then, the author launches one of the activities and interacts with one of the widgets in that activity. If that widget triggers a pop-up dialog box, the author chooses one option in the dialog box or just closes it and stores the test case. These are done randomly. If the option starts a new activity or does not have any other pop-up, we will directly stop. We repeat for each of the widgets in the activity to generate more test cases. Note that each time we start generating a new test case, we reinstall the app to avoid different initial conditions affecting the recorded test cases. As a result, for each feature, we generate four to six test cases based on the number of widgets within the target activity (214 in total). For each app, it typically takes 30 minutes to explore, generate, and record the test cases.

Finally, we use SARA to replay the user behaviors of instrumented apps and collect the executed methods while running.

5 EVALUATION SETUP

Here, we describe experiment settings and evaluation metrics.

5.1 Experiment Settings

MethodGeneralizer takes methods that are recorded during monitoring (i.e., m) as input, to identify methods of desired features (i.e., M'). To evaluate the performance of MethodGeneralizer, we randomly select the methods in one user behavior (except launching the app) as the methods in unseen user behavior(m_{test}, i.e., test set). Then, we use the methods in other user behaviors as the executed methods that are recorded during monitoring (m, i.e., input). We iteratively select the test set and executed methods m for each app until there are additional methods in the test set (i.e., $m_{test} \not\subset m$).

5.2 Evaluation Metrics

Specifically, we use recall, debloating rate, and a weighted harmonic mean of recall and debloating rate to evaluate the performance of MethodGeneralizer in identifying methods of desired features.

When users explore unseen behaviors (m_{test} would be executed), we would like to evaluate to what extent the methods in M' can satisfy users' needs, i.e., $m_{test} \subset M'$. This motivates us to use recall as one of the evaluation metrics. The recall of an approach is defined as the ability of an approach to identify the correct methods in M':

$$recall = \frac{|m_{test} \cap M'|}{|m_{test}|} \tag{1}$$

Intuitively, we want to maximize the recall value, in order to include methods of desired features that were unobserved before.

Similarly, the debloating rate is defined as the ability of an approach to remove the methods in M:

$$debloating\ rate = \frac{|M| - |M'|}{|M|} \tag{2}$$

Intuitively, we want to maximize the debloating rate, so that we can remove as many methods as possible. We do not consider the precision value, because we do not have the ground truth of the methods that are not related to the user-desired features (if we have, we can just remove these methods). Therefore, we cannot calculate the precision value and use debloating rate as a proxy of precision.

Note that there is a trade-off between recall and debloating rate. That is, identifying more methods that are related to the user-desired features may result in a lower debloating rate. In our work, we want to balance the recall and debloating rate.Among the two metrics, recall is more important than debloating rate, because we want to include all methods that are related to the user-desired features so that the app will not crash when using the debloated app. Following the definition of standard F_β score (i.e., $f_\beta = (1+\beta^2) \times \frac{precision \times recall}{\beta^2 \times precision + recall}$), we define the $F_{debloat}$ score as the weighted harmonic mean of recall and debloating rate. We set the β value to 1/2 so that we can give more weight to the recall value:

$$F_{debloat} = \frac{5 \times recall \times debloating\ rate}{4 \times recall + debloating\ rate} \tag{3}$$

A higher $F_{debloat}$ score means that we can achieve a higher recall value and a higher debloating rate at the same time, while we attach more importance to the recall value.

6 EVALUATION RESULT

6.1 How effective are different variants of MethodGeneralizer in identifying methods of desired features?

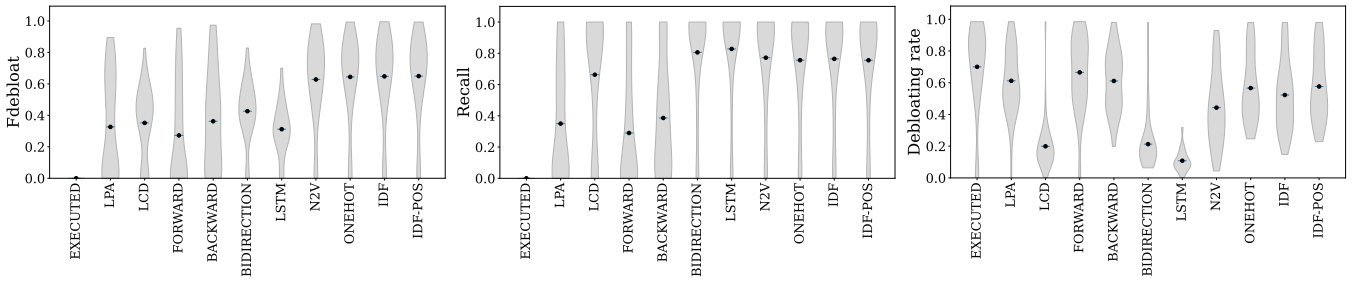
Motivation: We investigate whether MethodGeneralizer can effectively identify the additional methods of user-desired features that are exercised by unseen user behaviors.

Design: We evaluate and compare the performance of each MethodGeneralizer technique as well as a baseline without generalization (i.e., removes the unexecuted methods, shortened as EXECUTED) on our benchmark in terms of recall, debloating rate, and $F_{debloat}$.

Note that when using the graph embedding-based technique, MethodGeneralizer can generate different sets of M' and each of them corresponds to a similarity threshold. For each graph embedding-based technique, we select the M' set with the highest $F_{debloat}$ as the final result of this app. This version is the best version of the debloated app that can satisfy current users' needs for this new feature and minimize the number of methods to be kept.

Results: Figure 5 shows the $F_{debloat}$, recall, and debloating rate of each technique on each app. The dot · indicates the average value. As a result, we observe that graph embedding-based techniques (except LSTM) outperform all other techniques in terms of $F_{debloat}$, recall, and debloating rate on all the apps, and the differences are statistically significant with a large margin (p-value < 0.05 and effect sizes are large).[3] Besides, if we remove the unexecuted methods

[3]We use Wilcoxon signed-rank test to compare the performance of MiniMon variants and use the Bonferroni correction to adjust the p-value [33, 63]. We calculate Cliff's delta to measure the effect size [30]. Bonferroni correction is used to counteract the



(a) $F_{debloat}$ **(b) Recall** **(c) Debloating rate**

Figure 5: The distribution of $F_{debloat}$, recall, and debloating rate of each technique on each app. · indicates the average value.

during monitoring, the debloated apps cannot have the methods of user-desired features that are exercised by unseen user behaviors. However, there are no significant differences between the graph embedding-based techniques (except LSTM). The average $F_{debloat}$ of N2V, Onehot, IDF, and IDF-POS are 62.82%, 64.44%, 64.77%, and 64.95%, respectively. This indicates that these features do not have a significant impact on this task. However, considering the improvement in the average $F_{debloat}$, we take IDF-POS as the best methodGeneralizer technique.

We also investigate the worst case of IDF-POS, when the Cosine similarity threshold is 0.1. The average $F_{debloat}$, recall, and debloating rate are 52.89%, 77.24%, and 37.16%, resp., which is still better than techniques that are not based on graph embeddings. **This shows the advantage of graph embedding-based techniques on this task, no matter how nodes are embedded (whether use idf information or use word2vec). Future researchers should continue along this direction to generate better graph embedding-based representations of the methods in CG.**

Apart from graph embedding-based techniques, BIDIRECTION performs the best in terms of $F_{debloat}$ (42.64% on average). This indicates that simply involving all reachable methods in the CG is an intuitive but effective idea. However, BIDIRECTION involves too many methods that are not related to the user-desired features, which leads to a low debloating rate. In contrast, FORWARD achieves the worst performance. This indicates that **when determining whether two methods are similar, the methods that call these two methods are more important than the methods that are called by these two methods.**.

Apart from embedding-based techniques, LCD, the approach adopted in Tang et al's work, is the second-best technique in terms of $F_{debloat}$ (35.20% on average) with a high recall (66.30% on average). This indicates that the modularity of the app can effectively identify the methods related to user-desired features. However, modules identified by LCD are too coarse-grained, which leads to a low debloating rate. LPA performs worse than LCD in terms of $F_{debloat}$. This is because LPA only considers the local neighborhood, which may not reflect the correct method execution order.

LSTM is one of the worst-performing MethodGeneralizer variants in terms of $F_{debloat}$ and debloating rate. This is because LSTM aims to capture the pattern of the occurrence of methods in the static execution path, rather than the weight of each method in the execution path. Our result shows that **the pattern of methods occurrence in the static execution path has limited contribution when identifying similar methods**, as LSTM has the lowest debloating rate (keeps almost everything).

> Graph embedding-based generalization technique is the best, and can correctly uncover 75.5% of the unobserved but desired behaviors and still debloat more than half of the app.

6.2 Can MiniMon effectively debloat apps?

Motivation: Here, we would like to evaluate whether the debloated apps of the best technique can satisfy users' needs, i.e., can the debloated apps support the original and additional user behaviors? We also would like to understand the debloated apps generated by which series of MethodGeneralizers can satisfy users' needs.

Approach: To understand the benefit of debloating, in addition to the method level removal rate reported in Section 6.1, we would like to report the proportion of lines of decompiled code that are removed in the best-performing MethodGeneralizer (i.e., IDF-POS). To evaluate whether the debloated apps still can be used as usual, we use SARA to replay the interactions recorded in Section 4 on the debloated apps and compare the results with the original apps.

We also conduct a user study to investigate our approach's effectiveness. We advertise the user study to people in our institution through various means, e.g., internal chat channels. This process is similar to a prior study [35]. For each person who expressed interest, we present the 8 apps used in our user study and recruit only users who have used these apps (or other similar apps). 8 graduate students (5 Master students, and 3 Ph.D. students) participated in the user study. All of them are Android users with an average of more than 5 years of Android usage experience and have used all of these apps (or similar apps) before. The scale of our user study and the characteristics of our participants are similar to many prior studies [21, 28, 29, 34–36]. We first explained the concept of debloating to them, to ensure they understood the goal of their task. Then, we selected 8 apps and followed the procedure in Section 3 to instrument the apps. Finally, we have a total of $8 \times 8 = 64$ data points collected from real users, which is larger than the data points considered in the evaluation dataset used in the previous studies [43, 51]. We provide informative tutorials and instructions to help

problem of multiple comparisons. Cliff's delta is a non-parametric effect size measure to evaluate the amount of difference between two groups. Romano et al. define an absolute delta of less than 0.147, between 0.147 and 0.33, between 0.33 and 0.474 and above 0.474 as "Negligible", "Small", "Medium", "Large" effect size, respectively [30].

Table 1: The average scores obtained from our user study of different perspectives. For each perspective, the highest score is highlighted in bold while the lowest is highlighted with underlining. The higher the score, the higher the satisfaction.

	EXECUTED	BIDIRECTION	LCD	IPF-POS
Usability	3.46	3.89	3.93	**4.17**
Size	**4.30**	2.00	1.93	4.22
Generalization	2.05	2.89	3.00	**3.03**
Overall	3.35	2.98	3.02	**3.76**

users collect logs, and ask each user to use each app for 5 minutes in 3 days (5 minutes is longer than the typical average app session, i.e., 71.56 seconds [26]. This time interval is the same as in prior studies [21, 29]). On the first day, they are asked to freely explore the instrumented apps to understand the features of the apps. On the second and the third day, they are asked to use the instrumented apps with their desired features. Then we use the executed methods collected on the second day and the third day to debloat the apps using the 3 best-performing techniques in each family (i.e., LCD, BIDIRECTION, and IDF-POS). For IDF-POS, similarity thresholds are set to 0.3, 0.6, and 0.9 to minimize the human effort. Then, we ask users to use the debloated apps for 5 minutes (we show them the debloating rate so that they know which is the slimmer one). Users have no idea about how these techniques work, so they cannot figure out which debloated app is generated by which technique.

Finally, we ask each user to assign a score from 1 (very unsatisfied) to 5 (very satisfied) to each debloated app, in terms of (1) Usability (to what extent the debloated apps keep the desired features); (2) Generalization (to what extent the debloated apps allow them to perform behaviors that were not executed before but belong to the desired features); (3) Size (to what extent they are happy with the debloated apps' size reduction), and (4) Overall Satisfaction.

Results: We observe that the debloated apps have 38.8% fewer lines of decompiled code than the original apps on average. This indicates that our approach can effectively debloat apps in terms of the number of methods and lines of code omitted. Among all 214 test cases, 88.8% of the test cases succeeded to replay on the debloated apps generated by the best-performing MethodGeneralizer (i.e., IDF-POS) (24 test cases in 10 apps fail). We carefully investigate these 10 apps and find that there is no intersection between the static execution path of the methods in m and the methods in m_{test}. In this case, the similarity between these methods is 0, so they are not added to M'. One possible reason is that there can be flakiness when users use the app. Some of the methods are accidentally not triggered and thus cannot be recorded (e.g., turning off the Wi-Fi feature is not triggered when Wi-Fi is already turned off).

Table 1 shows the average scores assigned by the users. We also apply the statistics test described in Section 6.1. The results show that users have significantly higher satisfaction with the debloated apps generated by the IPF-POS in all aspects ($p < 0.05$ with non-negligible effective size) except the Size aspect. The use of the intelligent (generalization) method of MiniMon boosts the overall user satisfaction rate by 37.6%. In terms of Usability, users are the most satisfied with the debloated apps generated by the IPF-POS. Users are also satisfied with the debloated apps generated by the BIDIRECTION and LCD in terms of generalizing desired features ($p < 0.05$ with medium effect size, compared with EXECUTED). Users are least satisfied with the debloated apps generated by the BIDIRECTION and LCD in terms of the size of the debloated apps ($p < 0.05$ with large effect size, compared with EXECUTED and IPF-POS). In summary, the debloated apps generated by the IDF-POS are the most satisfying to the users.

Table 2: $F_{debloat}$, recall, and debloating rate of the variants of IDF-POS and N2V

	$F_{debloat}$	Recall	Debloating Rate
IDF-POS	**64.95%**	**75.52%**	**57.18%**
IDF-POS-PART	59.08%	69.67%	53.41%
IDF-POS-CLU	62.42%	75.12%	54.26%
N2V	**62.82%**	77.18%	**44.32%**
N2V-PART	44.87%	74.96%	27.24%
N2V-CLU	61.66%	**78.32%**	43.60%

88.8% of the test cases succeeded in replaying the debloated apps generated by the best-performing MethodGeneralizer. The use of the intelligent (generalization) method of MiniMon boosts the overall user satisfaction rate by 37.6%.

7 DISCUSSION

7.1 Analysis of the Best MethodGeneralizer

We analyze the effects of the main components in the best-performing MethodGeneralizer, i.e. the graph embedding-based techniques. Specifically, we would like to investigate IDF-POS (the best MethodGeneralizer), and the N2V. Other embedding-based techniques are variants of IDF-POS, and the N2V is a popular one.

Section 3.2.3 shows that the graph embedding-based techniques first (1) uses all nodes in CG to generate the embedding of the methods. Then it (2) directly considers two methods with similarity score higher than a specific threshold to be similar. However, we do not know which component is the most important. We, therefore, investigate the effects of each component in MiniMon.

We compare each MethodGeneralizer with its 2 variants:

- **MethodGeneralizer-PART** uses one-third (rather than all) of the nodes in the neighbor in each static execution path to generate the embedding of the methods.
- **MethodGeneralizer-CLU** uses hierarchical clustering to identify similar methods (rather than directly adding the nearest neighbors with similarity score between them higher than the threshold). Each hierarchy with a certain similarity level (following the settings in Section 3, the similarity levels are set from 0 to 1 with step 0.1) hosts a set of clusters. For each hierarchy, if there are executed methods in the cluster, we would add all the methods in the cluster to M'.

Such comparisons can help us understand the impacts of the components on MethodGeneralizer's performance.

Table 2 shows the results of our comparison. We see that IDF-POS and N2V outperform all their variants in terms of $F_{debloat}$ and debloating rate. This indicates that IDF-POS and N2V can both effectively identify the methods of unseen user behavior in desired features and remove the methods of undesired features.

IDF-POS and N2V outperform their MethodGeneralizer-PART variants indicating that **global information of the graph (i.e., considering all nodes in the graph) is the most important component in identifying the methods that are related to the user-desired features both in IDF-POS and N2V**. Considering two *onclick*() methods in two different UI components, they invoke different methods, but after a long chain in CG, both eventually call FTP-related methods. If only the local information in CG is considered, these two methods are not similar because they invoke different methods. However, if the entire call information is taken into account, these two *onclick*() methods are similar because they eventually call FTP-related methods. Furthermore, when we only consider one-third of the nodes in the neighbor in each static execution path, the performance drops significantly, especially in N2V. This indicates that **when the window size in Node2Vec cannot cover the whole graph, the performance of N2V drops significantly**. Future researchers should take all nodes (with a large enough window size) in CG to generate the embedding of methods.

Moreover, the clustering method is not as effective as simply adding the nearest neighbors to M'. One possible reason is that the clustering methods identify all the methods in the cluster as similar methods, which may introduce more noise.

7.2 Limitations and Threats to Validity

The findings in this paper may not generalize to all apps. For example, similar to prior work [43, 59], we have not considered the apps with tampering detection mechanisms and the features implemented in native code. We cannot modify apps with tampering detection mechanisms, because these apps would detect the modification and stop working [5]. Besides, we focus on dex files in this paper. The features implemented in native code are not available in dex files and need additional effort to analyze and instrument. We believe that once we obtain the CG of these applications, the MethodGeneralizer component of our work can also be generalized to these applications. In the future, we plan to evaluate our work on more apps and investigate how to debloat the apps with tampering detection mechanisms and the features implemented in native code.

We employ some heuristics when we are building the benchmark. For example, our benchmark is created by authors with an assumption that widgets in the same activity are likely to be related to the same feature (based on guidelines given in [11, 69]), rather than by real users. Still, there are likely to be apps that do not follow these guidelines. We acknowledge that the benchmark creation process is not perfect, yet, there is no other alternative benchmark. Even the current benchmark that we create requires much manual work. To mitigate the threats from an imperfect benchmark, we evaluate our proposed approach through a user study, where we collect real users' interactions with apps.

Threats to construct validity relate to the evaluation metrics that we use. We use debloating rate, recall, and $F_{debloat}$ to evaluate our approach. We do not report the actual app size reduction because the resources in each app take up a lot of space, and we focus on removing the code (i.e., methods) in this paper. We perform experiments with real-world apps on a real device by humans. The second threat is related to the limitations of the user study. We only have 8 participants in the user study. However, this study size is also followed by prior work [21, 28, 29, 34–36]. These reduce the threat to construct validity.

8 RELATED WORK

Debloating Android apps is a known topic in the Android community. Google proposed approaches to reduce the size of Android apps. For example, developers can (1) use R8 to statically detect and remove dead code and its library dependencies, (2) remove unused resources, and (3) shorten the names of classes and members to reduce the DEX file sizes [12]. Developers can also configure App Bundle or use the on-demand delivery so that only the code and resources that are needed for a specific device or feature are downloaded [2, 8]. These approaches have become the default settings for Android app development, and are complementary to MiniMon. MiniMon can be run before or after R8. We leveraged the settings from Bruce et al.'s work [27], to check whether the apps in our experiments could be further debloated by these methods. Bruce et al. compared their approach (i.e., JShrink) with ProGuard (using the same settings as R8) and other static analysis-based approaches. We found that no methods could be removed. This means that developers are already using static tools like R8 to debloat Android apps during the build phase. Despite this, our study shows that there are still a large number of methods that a specific user does not need in these apps, and our work is needed for better user-specific app debloating.

Researchers proposed approaches to reduce Android apps' size. Xie et al. debloated apps to minimize the bandwidth of mobile networks [64]. Jiang et al. removed dead code based on static analysis [44]. Tang et al. debloated apps at the granularity of Activity, Permission, and Modularity [59]. To debloat apps at the Modularity level, they considered each module to be a feature.They used Louvain community detection to identify the modules in CG. Following their work, we also use LCD as one of our techniques. Our results show that LCD is inferior to the other graph embedding-based techniques in identifying the methods related to user-desired features.

Pilgun et al. removed the unexecuted instruction during testing [51]. We try to compare MiniMon with Pilgun's work [51] at the instruction level, but faced difficulties (we can only replicate their results on two examples apps). Though being in contact with the authors, we were unable to fix the issues. Pilgun promised to run their tool on more apps but has not delivered when we submitted our paper. Therefore, we implement a baseline that removes the unexecuted methods (i.e., EXECUTED). Our results show that MiniMon outperforms EXECUTED in terms of $F_{debloat}$ and Recall.

Huang et al. proposed a method to remove unused UI components [43]. They asked users to specify components to remove and use forward/backward slicing to identify instructions related to UI components. They eliminated instructions that are only relevant to the relevant components. However, our work aims to preserve all executed methods and generalize them to user behaviors in the future. Unlike Huang et al., we do not know which UI components are unnecessary. Therefore, our approach and Huang et al.'s approach have different goals, and we cannot directly compare our method to theirs. Nevertheless, we believe that the EXECUTED baseline method, which records log information from the instrumented app to identify the methods related to UI components, to some extent

embodies Huang et al.'s approach to finding UI component-related methods. Additionally, our program analysis-based techniques also partially leverage Huang et al.'s approach in terms of backward slicing and forward slicing. We are confident that we have made the most relevant comparisons to Huang et al.'s method.

Qian et al. [53] and Xin et al. [66] proposed approaches to debloat C/C++ programs, e.g., UNIX utilities, based on the usage profile of an individual user. They first collect the log of statements executed when a C/C++ program is used. Then, they employ some augmentation techniques to identify program elements that are related to the executed one. The executed and related program statements are kept, while the rest are deleted. Although the motivation of our work and that of the two studies are similar, there are differences that make the works complementary. First, the augmentation techniques employed by Qian et al. and Xin et al. require instrumentation at the statement level. On the other hand, our proposed approach (MiniMon) only requires instrumentation at the method level, which is more lightweight and results in performance overhead that is acceptable to Android phone users. Moreover, due to the different levels of instrumentation considered, heuristics employed in the augmentation techniques of the two papers cannot be easily applied to MiniMon. For example, Qian et al.'s augmentation techniques include 4 strategies: adding additional branches, as well as reachable instructions, reachable functions within the same binary or executed external functions, and reachable library functions with the same functionalities. It is unclear how these heuristics can be applied to MiniMon to identify related methods. We believe that the FORWARD technique, which considers all invoked methods as the methods of user-desired features, to some extent embodies Qian et al.'s approach. Also, the computation of flexibility in Xin et al.'s augmentation technique (CovA) requires the counting of the number of unique sets of executed statements in a C/C++ function.

Our work is also related to feature location. Existing works primarily focus on facilitating program comprehension [32, 55]. In our work, we use the features that are collected during monitoring, to find the methods corresponding to all user-desired features. We believe that many of the works in the feature location can be adapted to our task. We have already compared our approach with some feature location approaches used in our field [59]. In the future, we plan to explore other feature location works (if based on unsupervised learning) to further enhance the performance.

9 CONCLUSION AND FUTURE WORK

In this paper, we aim to debloat Android apps based on how users use them. To address this task, we implement a monitoring-based debloating framework MiniMon that (1) can collect the executed methods when monitoring. Then, together with the app call graph, the MethodGeneralizer component in MiniMon (2) adopts three program analysis-based techniques, two graph clustering-based techniques, and five graph embedding-based techniques to identify additional methods of desired features. Finally, MiniMon (3) generates the debloated apps by removing the remaining methods. For evaluation, we manually create a benchmark that collects the methods executed by each feature. Controlled experiments using this benchmark highlight that the embedding-based generalization technique that considers the information of all nodes in the call graph is the best, and can correctly uncover 75.5% of additional methods of desired features and debloat more than half of the app. Our user study that using the intelligent (generalization) method of MiniMon boosts the overall user satisfaction rate by 37.6%.

In the future, we plan to other Android-specific characteristics to identify the methods related to the user's desired features. We also plan to enlarge the benchmark using more apps and test cases.

DATA AVAILABILITY

MiniMon's replication package, including the dataset and the source code, is publicly available at https://zenodo.org/doi/10.5281/zenodo.8201782.

ACKNOWLEDGEMENTS

This research / project is supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity Research and Development Programme, NCRP25-P03-NCR-TAU. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore. This work is also supported by the Blavatnik Interdisciplinary Cyber Research Center at Tel Aviv University as part of a collaboration with CSA Singapore.

REFERENCES

[1] 2019. Dynatrace Android Gradle Plugin. https://www.dynatrace.com/support/help/platform-modules/digital-experience/mobile-applications/instrument-android-app/instrumentation-via-plugin
[2] 2023. About Android App Bundles. https://developer.android.com/guide/app-bundle
[3] 2023. Amaze File Manager - Apps on Google Play. https://play.google.com/store/apps/details?id=com.amaze.filemanager&hl=en
[4] 2023. Android. https://frida.re/docs/android/
[5] 2023. Android Anti-Reversing Defenses. https://mobile-security.gitbook.io/mobile-security-testing-guide/android-testing-guide/0x05j-testing-resiliency-against-reverse-engineering
[6] 2023. Android Mobile App Developer Tools. https://developer.android.com/
[7] 2023. BodyTransformer (Soot API). https://www.sable.mcgill.ca/soot/doc/soot/BodyTransformer.html
[8] 2023. Configure on Demand Delivery. https://developer.android.com/guide/playcore/feature-delivery/on-demand
[9] 2023. Label_propagation_communities — NetworkX 3.1 Documentation. https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.community.label_propagation.label_propagation_communities.html#networkx.algorithms.community.label_propagation.label_propagation_communities
[10] 2023. Louvain_partitions — NetworkX 3.1 Documentation. https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.community.louvain.louvain_partitions.html#networkx.algorithms.community.louvain.louvain_partitions
[11] 2023. Material Design. https://m2.material.io/design/layout/understanding-layout.html#layout-anatomy
[12] 2023. Shrink, Obfuscate, and Optimize Your App | Android Studio. https://developer.android.com/build/shrink-code
[13] 2023. Twitter 9.86.0-Release.0 Build Variants in Android - APK Download. https://apkpure.com/twitter/com.twitter.android/variant/9.86.0-release.0-XAPK
[14] 2023. ViewGroup. https://developer.android.com/reference/android/view/ViewGroup
[15] Adil Aijaz and Carol Jang. 2020. The 80% Rule of Software Development. https://www.split.io/blog/the-80-rule-of-software-development/
[16] Rakan Alanazi, Gharib Gharibi, and Yugyung Lee. 2021. Facilitating Program Comprehension with Call Graph Multilevel Hierarchical Abstractions. *Journal of Systems and Software* 176 (June 2021), 110945. https://doi.org/10.1016/j.jss.2021.110945

[17] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. 2018. Inferring Hierarchical Motifs from Execution Traces. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg Sweden, 776–787. https://doi.org/10.1145/3180155.3180216

[18] Amplitude. 2023. Android Kotlin SDK - Amplitude Developer Center. https://docs.developers.amplitude.com/data/sdks/android-kotlin/

[19] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Edinburgh United Kingdom, 259–269. https://doi.org/10.1145/2594291.2594299

[20] F Asadi, M Di Penta, G Antoniol, and Yann-Gaël Guéhéneuc. 2010. A Heuristic-Based Approach to Identify Concepts in Execution Traces. In *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, Madrid, 31–40. https://doi.org/10.1109/CSMR.2010.17

[21] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and Depth-First Exploration for Systematic Testing of Android Apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. ACM, Indianapolis Indiana USA, 641–660. https://doi.org/10.1145/2509136.2509549

[22] Suparna Bhattacharya, Kanchi Gopinath, and Mangala Gowri Nanda. 2013. Combining Concern Input with Program Analysis for Bloat Detection. *ACM SIGPLAN Notices* 48, 10 (Nov. 2013), 745–764. https://doi.org/10.1145/2544173.2509522

[23] T.J. Biggerstaff, B.G. Mitbander, and D. Webster. 1993. The Concept Assignment Problem in Program Understanding. In *[1993] Proceedings Working Conference on Reverse Engineering*. IEEE Comput. Soc. Press, Baltimore, MD, USA, 27–43. https://doi.org/10.1109/WCRE.1993.287781

[24] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, Hong Kong China, 30–40. https://doi.org/10.1145/1966913.1966919

[25] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast Unfolding of Communities in Large Networks. *Journal of Statistical Mechanics: Theory and Experiment* 2008, 10 (Oct. 2008), P10008. https://doi.org/10.1088/1742-5468/2008/10/P10008

[26] Matthias Böhmer, Brent Hecht, Johannes Schöning, Antonio Krüger, and Gernot Bauer. 2011. Falling Asleep with Angry Birds, Facebook and Kindle: A Large Scale Study on Mobile Application Usage. In *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*. ACM, Stockholm Sweden, 47–56. https://doi.org/10.1145/2037373.2037383

[27] Bobby R. Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. 2020. JShrink: In-Depth Investigation into Debloating Modern Java Applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Virtual Event USA, 135–146. https://doi.org/10.1145/3368089.3409738

[28] Jieshan Chen, Jiamou Sun, Sidong Feng, Zhenchang Xing, Qinghua Lu, Xiwei Xu, and Chunyang Chen. 2023. Unveiling the Tricks: Automated Detection of Dark Patterns in Mobile Applications. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. ACM, San Francisco CA USA, 1–20. https://doi.org/10.1145/3586183.3606783

[29] Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenhe Li, Yang Liu, and Lihua Xu. 2019. StoryDroid: Automated Generation of Storyboard for Android Apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, Montreal, QC, Canada, 596–607. https://doi.org/10.1109/ICSE.2019.00070

[30] Norman Cliff. 2014. *Ordinal Methods for Behavioral Data Analysis* (0 ed.). Psychology Press. https://doi.org/10.4324/9781315806730

[31] Elior Cohen. 2023. Node2Vec. https://github.com/eliorc/node2vec

[32] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature Location in Source Code: A Taxonomy and Survey: FEATURE LOCATION IN SOURCE CODE: A TAXONOMY AND SURVEY. *Journal of Software: Evolution and Process* 25, 1 (Jan. 2013), 53–95. https://doi.org/10.1002/smr.567

[33] Olive Jean Dunn. 1961. Multiple Comparisons among Means. *J. Amer. Statist. Assoc.* 56, 293 (March 1961), 52–64. https://doi.org/10.1080/01621459.1961.10482090

[34] Sidong Feng and Chunyang Chen. 2022. GIFdroid: Automated Replay of Visual Bug Reports for Android Apps. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh Pennsylvania, 1045–1057. https://doi.org/10.1145/3510003.3510048

[35] Sidong Feng and Chunyang Chen. 2023. Prompting Is All You Need: Automated Android Bug Replay with Large Language Models. arXiv:2306.01987 [cs] http://arxiv.org/abs/2306.01987

[36] Sidong Feng, Mulong Xie, Yinxing Xue, and Chunyang Chen. 2023. Read It, Don't Watch It: Captioning Bug Recordings Automatically. arXiv:2302.00886 [cs] http://arxiv.org/abs/2302.00886

[37] Yang Feng, Kaj Dreef, James A. Jones, and Arie Van Deursen. 2018. Hierarchical Abstraction of Execution Traces for Program Comprehension. In *Proceedings of the 26th Conference on Program Comprehension*. ACM, Gothenburg Sweden, 86–96. https://doi.org/10.1145/3196321.3196343

[38] Masoud Ghaffarinia and Kevin W. Hamlen. 2019. Binary Control-Flow Trimming. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1009–1022. https://doi.org/10.1145/3319535.3345665

[39] Aditya Grover and Jure Leskovec. 2016. Node2vec: Scalable Feature Learning for Networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, San Francisco California USA, 855–864. https://doi.org/10.1145/2939672.2939754

[40] Jiaqi Guo, Shuyue Li, Jian-Guang Lou, Zijiang Yang, and Ting Liu. 2019. Sara: Self-Replay Augmented Record and Replay for Android in Industrial Cases. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Beijing China, 90–100. https://doi.org/10.1145/3293882.3330557

[41] Emily Hill, Lori Pollock, and K. Vijay-Shanker. 2007. Exploring the Neighborhood with Dora to Expedite Software Maintenance. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*. ACM, Atlanta Georgia USA, 14–23. https://doi.org/10.1145/1321631.1321637

[42] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[43] Jianjun Huang, Yousra Aafer, David Perry, Xiangyu Zhang, and Chen Tian. 2017. UI Driven Android Application Reduction. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Urbana, IL, 286–296. https://doi.org/10.1109/ASE.2017.8115642

[44] Yufei Jiang, Qinkun Bao, Shuai Wang, Xiao Liu, and Dinghao Wu. 2018. RedDroid: Android Application Redundancy Customization Based on Static Analysis. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. 189–199. https://doi.org/10.1109/ISSRE.2018.00029

[45] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. arXiv:1609.02907 [cs, stat] http://arxiv.org/abs/1609.02907

[46] Ziqi Liu, Chaochao Chen, Longfei Li, Jun Zhou, Xiaolong Li, Le Song, and Yuan Qi. 2018. GeniePath: Graph Neural Networks with Adaptive Receptive Paths. arXiv:1802.00910 [cs] http://arxiv.org/abs/1802.00910

[47] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-Objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, Saarbrücken Germany, 94–105. https://doi.org/10.1145/2931037.2931054

[48] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, Florence, Italy, 77–88. https://doi.org/10.1109/ICSE.2015.30

[49] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective {Inter-Component} Communication Mapping in Android: An Essential Step Towards Holistic Security Analysis. In *22nd USENIX Security Symposium (USENIX Security 13)*. 543–558. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/octeau

[50] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: Online Learning of Social Representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, New York New York USA, 701–710. https://doi.org/10.1145/2623330.2623732

[51] Aleksandr Pilgun. 2020. Don't Trust Me, Test Me: 100% Code Coverage for a 3rd-Party Android App. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, Singapore, Singapore, 375–384. https://doi.org/10.1109/APSEC51365.2020.00046

[52] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. 2020. BlankIt Library Debloating: Getting What You Want Instead of Cutting What You Don't. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 164–180. https://doi.org/10.1145/3385412.3386017

[53] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. {RAZOR}: A Framework for Post-deployment Software Debloating. In *28th USENIX Security Symposium (USENIX Security 19)*. 1733–1750. https://www.usenix.org/conference/usenixsecurity19/presentation/qian

[54] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '95*. ACM Press, San Francisco, California, United States, 49–61. https://doi.org/10.1145/199448.199462

[55] Julia Rubin and Marsha Chechik. 2013. A Survey of Feature Location Techniques. In *Domain Engineering: Product Lines, Languages, and Conceptual Models*, Iris Reinhartz-Berger, Arnon Sturm, Tony Clark, Sholom Cohen, and Jorn Bettin (Eds.). Springer, Berlin, Heidelberg, 29–58. https://doi.org/10.1007/978-3-642-36654-3_2

[56] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. 2008. Collective Classification in Network Data. *AI Magazine*

29, 3 (Sept. 2008), 93. https://doi.org/10.1609/aimag.v29i3.2157

[57] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*. ACM, Alexandria Virginia USA, 552–561. https://doi.org/10.1145/1315245.1315313

[58] Peng Shao and Randy K. Smith. 2009. Feature Location by IR Modules and Call Graph. In *Proceedings of the 47th Annual Southeast Regional Conference*. ACM, Clemson South Carolina, 1–4. https://doi.org/10.1145/1566445.1566539

[59] Yutian Tang, Hao Zhou, Xiapu Luo, Ting Chen, Haoyu Wang, Zhou Xu, and Yan Cai. 2022. XDebloat: Towards Automated Feature-Oriented App Debloating. *IEEE Transactions on Software Engineering* 48, 11 (Nov. 2022), 4501–4520. https://doi.org/10.1109/TSE.2021.3120213

[60] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2017. Graph Attention Networks. (2017). https://doi.org/10.48550/ARXIV.1710.10903

[61] Neil Walkinshaw, Marc Roper, and Murray Wood. 2007. Feature Location and Extraction Using Landmarks and Barriers. In *2007 IEEE International Conference on Software Maintenance*. IEEE, Paris, 54–63. https://doi.org/10.1109/ICSM.2007.4362618

[62] Mark Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4 (July 1984), 352–357. https://doi.org/10.1109/TSE.1984.5010248

[63] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (Dec. 1945), 80. https://doi.org/10.2307/3001968 jstor:10.2307/3001968

[64] Qinge Xie, Qingyuan Gong, Xinlei He, Yang Chen, Xin Wang, Haitao Zheng, and Ben Y. Zhao. 2023. Trimming Mobile Applications for Bandwidth-Challenged Networks in Developing Regions. *IEEE Transactions on Mobile Computing* 22, 1 (Jan. 2023), 556–573. https://doi.org/10.1109/TMC.2021.3088121

[65] Qi Xin, Farnaz Behrang, Mattia Fazzini, and Alessandro Orso. 2019. Identifying Features of Android Apps from Execution Traces. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, Montreal, QC, Canada, 35–39. https://doi.org/10.1109/MOBILESoft.2019.00015

[66] Qi Xin, Qirun Zhang, and Alessandro Orso. 2022. Studying and Understanding the Tradeoffs Between Generality and Reduction in Software Debloating. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Rochester MI USA, 1–13. https://doi.org/10.1145/3551349.3556970

[67] Chunyan Xu, Zhen Cui, Xiaobin Hong, Tong Zhang, Jian Yang, and Wei Liu. 2020. Graph Inference Learning for Semi-supervised Classification. arXiv:2001.06137 [cs, stat] http://arxiv.org/abs/2001.06137

[68] Jiwei Yan, Shixin Zhang, Yepang Liu, Jun Yan, and Jian Zhang. 2022. ICCBot: Fragment-Aware and Context-Sensitive ICC Resolution for Android Applications. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. ACM, Pittsburgh Pennsylvania, 105–109. https://doi.org/10.1145/3510454.3516864

[69] Bo Yang, Zhenchang Xing, Xin Xia, Chunyang Chen, Deheng Ye, and Shanping Li. 2021. Don't Do That! Hunting Down Visual Design Smells in Complex UIs Against Design Guidelines. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, Madrid, ES, 761–772. https://doi.org/10.1109/ICSE43902.2021.00075

[70] Seongjun Yun, Minbyul Jeong, Raehyun Kim, Jaewoo Kang, and Hyunwoo J Kim. 2019. Graph Transformer Networks. In *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/hash/9d63484abb477c97640154d40595a3bb-Abstract.html

[71] Xiaojin Zhu and Zoubin Ghahramani. 2002. Learning from Labeled and Unlabeled Data with Label Propagation. (2002).