9-2023

Fine-grained in-context permission classification for Android apps using control-flow graph embedding

Vikas Kumar MALVIYA
Singapore Management University, vikasm@smu.edu.sg

Naing Tun YAN
Singapore Management University, yannaingtun@smu.edu.sg

Chee Wei LEOW
Singapore Management University, cwleow@smu.edu.sg

Ailys Xynyn TEE
Singapore Management University, ailystee.2020@scis.smu.edu.sg

Lwin Khin SHAR
Singapore Management University, lkshar@smu.edu.sg

See next page for additional authors

Fine-Grained In-Context Permission Classification for Android Apps using Control-Flow Graph Embedding

Vikas K. Malviya
Singapore Management University
Singapore
vikasm@smu.edu.sg

Yan Naing Tun
Singapore Management University
Singapore
yannaingtun@smu.edu.sg

Chee Wei Leow
Singapore Management University
Singapore
cwleow@smu.edu.sg

Ailys Tee Xynyn
Singapore Management University
Singapore
ailystee.2020@scis.smu.edu.sg

Lwin Khin Shar
Singapore Management University
Singapore
lkshar@smu.edu.sg

Lingxiao Jiang
Singapore Management University
Singapore
lxjiang@smu.edu.sg

***Abstract*—Android is the most popular operating system for mobile devices nowadays. Permissions are a very important part of Android security architecture. Apps frequently need the users' permission, but many of them only ask for it once—when the user uses the app for the first time—and then they keep and abuse the given permissions. Longing to enhance Android permission security and users' private data protection is the driving factor behind our approach to explore fine-grained context-sensitive permission usage analysis and thereby identify misuses in Android apps. In this work, we propose an approach for classifying the fine-grained permission uses for each functionality of Android apps that a user interacts with. Our approach, named DROIDGEM, relies on mainly three technical components to provide an in-context classification for permission (mis)uses by Android apps for each functionality triggered by users: (1) static inter-procedural control-flow graphs and call graphs representing each functionality in an app that may be triggered by users' or systems' events through UI-linked event handlers, (2) graph embedding techniques converting graph structures into numerical encoding, and (3) supervised machine learning models classifying (mis)uses of permissions based on the embedding. We have implemented a prototype of DROIDGEM and evaluated it on 89 diverse apps. The results show that DROIDGEM can accurately classify whether permission used by the functionality of an app triggered by a UI-linked event handler is a misuse in relation to manually verified decisions, with up to 95% precision and recall. We believe that such a permission classification mechanism can be helpful in providing fine-grained permission notices in a context related to app users' actions, and improving their awareness of (mis)uses of permissions and private data in Android apps.**

***Index Terms*—Privacy protection, Permission control, Android apps, Control flow graphs, Graph embedding, Classification.**

I. INTRODUCTION

Smartphones and other mobile devices are being used extensively in daily life. Among various operating systems, Android is the most popular operating system today with 68.79% of market share in April 2023 [1]. Due to its popularity, apps for everyday chores are available for the Android platform. Concerns related to user privacy are also growing along with the adoption of Android as some apps use users' data without their knowledge for advertising or other detrimental purposes.

Android platform uses a permission control mechanism to enforce restrictions on the operations of apps. In this way, Android security architecture ensures that no application can impact other applications, operating systems, or users (which includes users' private data) without suitable permissions. While an app is running, it has to request permissions from users to access sensitive data or services controlled by the permissions on the device. However, there are various kinds of deficiencies associated with the current permission control mechanism in Android. For example, the request for permission is often done only once when the app tries to access the private data for the first time; once the user grants permission, the app retains the permission and may continue to use or abuse it for different purposes without informing the user. Further, many apps do not explain clearly the actual purposes of their permission requests; most users are not aware of the effects of granting such permissions and they blindly grant the permissions most of the time. These flaws in the permission control mechanism easily result in violations of the right to know for users and place an undue burden on their cognitive abilities without giving users enough notice to raise their awareness (Hong 2017) [2], causing misuse of users' private data even by benign apps. The reputation of app developers is also not a reliable means to trust the apps either. For example, Facebook (Constine 2019, Shealy 2019) [3], [4] and Google (Whittaker et al. 2019) [5] were found to collect user data without users' knowledge.

Despite the fact that benign apps typically implement a Notice-and-Choice mechanism for users when requesting permissions [6], the notices and choices given to users are often too *coarse-grained* for the entire app instead of individual functionality, *untimely* when the user uses the app for the first time instead of when the user uses a specific functionality of the app that really needs the permission and private data for the functionality, and *out-of-context* where only vague summaries are given instead of clear explanation of the purposes of using the private data for the functionality. Along with other limitations, these limitations of Notice-and-Choice have

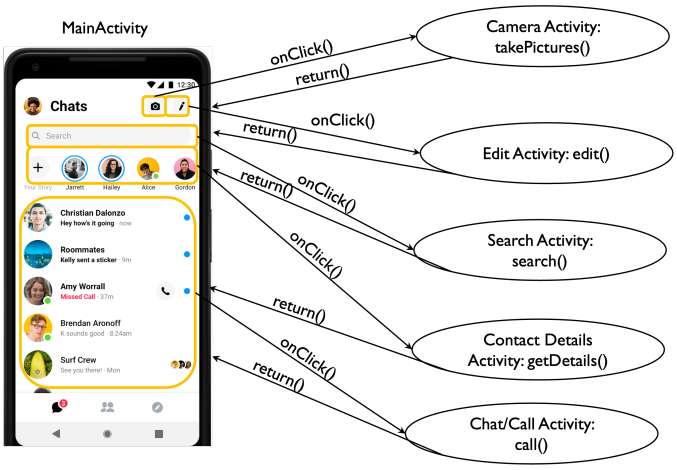

Fig. 1. Main Idea of Approach

prompted improvements and alternate mechanisms to have more expressive, detailed, machine-readable, automated, and customizable privacy notices and choices [7].

This paper proposes a new approach for classifying permission uses and abuses in Android apps associated with each functionality that a user interacts with. The classifications can help users decide whether or not to grant an Android app permission by demonstrating how that permission is required for a feature that the user uses.

As an illustration, Figure 1 shows the main idea of our approach. On the left side of the figure, the `MainActivity` represents a screen of an app. Different UI elements shown on the screen are linked by various event handlers (e.g., `onClick()` functions) to different functionalities in different activities (indicated by the ovals on the right side of the figure). For example, when a user taps on the camera icon on the screen, the `takePicture()` function of the `CameraActivity` can be invoked through the event handler. Since the camera is a system resource in Android, the CAMERA permission will be needed by this function, and the use of the permission can be detected in the underlying code triggered by the `onClick` and `takePicture` functions. Also, by capturing all of the code executed from the UI event handlers to the functions triggered until their returns in a certain form, we expect that we can classify whether the CAMERA permission is indeed needed by the functionality related to the UI event and functionality, based on certain domain knowledge about common uses of the permissions and machine learning models. In the case of the `onClick` for the camera icon, our classification indicates that it is benign for the code to use the CAMERA permission for taking photos in responding to the event handler for the camera icon, consistent with the actual permission used in the code. Similarly, when a user taps on one of the contacts in the list, the `onClick` event handler for the list and the associated `getDetails()` function in the `ContactDetails` Activity will be triggered. The underlying code for the functions will show that the code needs to use the READ_CONTACTS permission, and our classification also indicates the use is benign for the UI event and functionality for getting the details of a contact. On the other hand, if the underlying code executed for one of the triggered functions shows that the code uses the ACCESS_FINE_LOCATION permission but our classification indicates that the permission may not be needed for that UI event and the functionality of the event does not need to read the user's location, thereby the actual permission used in the code would be classified as misuse. Thus, our approach can help in achieving per-functionality permission classification in accordance with each user action. The classification results can also highlight different permissions used inappropriately in the app code and the permission(s) required for the functionality associated with the user action.

This paper realizes our approach based on mainly three technical components. First, we need a representation of each functionality in an app that a user can interact with. Android apps consist of graphical user interface (GUI) resources and bytecodes. There are event handlers (e.g., `onClick()` functions) attached to UI elements and can be triggered by various kinds of events from users and/or the Android systems to perform certain functionality. This behaviour can be represented with inter-procedural control-flow graphs (ICFGs) and call graphs using static program analysis (Section III-B). Second, we need to process the graph representations of app functionalities in a way that can be easily used to classify its behaviours and permission (mis)uses. We adopt graph embedding techniques for this purpose, converting the various code graphs to unified numerical vectors (Section III-C), which can then be used to train various machine-learning models. Third, we must identify whether each functionality a user interacts with is using permission in a benign or malicious manner. We construct a set of training data based on a combination of automated and manual analysis (Section III-D) and adapt supervised machine learning models (Section III-E) to classify the permission (mis)uses based on the graph embedding representing each functionality.

Our approach demonstrates that it is feasible to identify the permission needs of a piece of code, linked to Android event handlers and user actions. It builds the basis for strengthening the broader idea that inconsistencies between app user interfaces and underlying code can be used to detect permission misuses and privacy violations. Our key technical contributions in this paper include:

- A static analysis-based technique for Android apps to construct inter-procedural control flow graphs (ICFGs) with call relations that can be used to represent the functionality of event handlers.
- A graph embedding technique for ICFGs to encode various code graph structures.
- A curated set of training data containing benign and malicious permission uses, and trained machine learning models based on the graph embedding that is able to differentiate malicious and benign permission uses with high accuracy.

Our evaluation of a prototype implementation of our ap-

proach, named DROIDGEM, using 89 real-world apps, shows promising results. DROIDGEM identifies malicious and benign uses of permissions with respect to the functionality triggered by a UI-linked event handler with a high F1-score of 95%. We believe that precise permission (mis)uses classification in the context of user actions, can enhance privacy protection and raise knowledge of the (mis)uses of permissions and personal data in Android apps.

In the rest of the paper, Section II surveys related work and discusses the uniqueness of our approach. Section III explains our approach. Section IV presents empirical evaluation and results. Section V concludes with future work.

II. RELATED WORK

A. Understanding and Managing Privacy

Numerous studies are proposed to provide users with the knowledge and skills needed to understand and control their privacy more effectively. In order to make privacy and security control easier for users, the CHIMPS (Computer-Human Interaction: Mobility Privacy Security) lab at CMU advocates for an ecosystem of privacy that transfers responsibility from end-users to other entities [2]. The objectives of data sent from a user's device through network traffic are determined by MobiPurpose, a tool developed by Jin et al. [8]. Chitkara et al [9] demonstrated that per-app authorization is too coarse-grained and that various intentions of private data access in an app may be deduced. Our work is motivated by the similar idea that more fine-grained context-aware permission/privacy management mechanisms are needed, although our approach to realizing the idea is very different.

B. Detection of Abnormal Uses of APIs, Permissions, and Private Data

Many studies have aimed to detect misuse of Android APIs, permissions, and private data using various techniques. A functional programming model called PrivacyStreams is introduced by Li et al. [10] for enabling developers to access private data and make it simpler to determine how data is actually used. CHABADA by Gorla et al. [11] uses the app descriptions to determine whether the apps' uses of sensitive APIs are unusual. MUDFLOW by Avdiienko et al. [12] analyses data flow to create classification models that distinguish between normal and improper uses of private data. BOXMATE by Jamrozik et al. [13] infers sandbox rules to limit personal data uses that take place in real runs but go unnoticed during testing. "GUI mining" is utilized by Avdiienko et al. [14] and Hotzkow [15] to group comparable GUI elements across apps and find outliers that call various APIs in the code. VetDroid by Zhang et al. [16] uses dynamic analysis to identify permission (mis)uses in apps for malware detection. Fu et al. [17] create classifiers that distinguish between genuine and erroneous authorization uses using app code and foreground user interfaces. 32 different types of context information, such as time, location, and app name, are employed in SmarPer by Olejnik et al. [18] to create classification models. Different from our approach, those studies have not taken fine-grained permission legitimacy for each functionality associated with an event handler linked to user actions into account.

C. User Perception versus App Behaviour

Many studies also utilize the (in)consistencies between users' perceptions of app behaviours and the actual app behaviours for detecting abnormal behaviour and/or permission misuses in the apps. Permission Event Graphs are used by Chen et al. in Pegasus [19] to describe policies on the interactions between user actions and permissions. The policies are then enforced using static analysis, model checking, and runtime monitoring for malware detection. App functionalities are employed in DroidJust by Chen and Zhu [20] to support whether a data transmission is a leak or not. AppIntent by Yang et al. [21] leverages GUI change event sequences to determine whether a data transmission is what the user intended. A similar issue is addressed in FlowIntent by Fu et al. [22] but with more automated data flow information. To prevent transferring sensitive on-screen data off the device, Appstract by Fernandes et al. [23] infers semantic annotations for UI elements on-device. AppContext by Yang et al. [24] employs circumstances and events surrounding private data usage to distinguish between malicious and benign uses. Both Whyper by Pandita et al. [25] and AutoCog by Qu et al. [26] examine whether an app's descriptions match the uses for which it has been granted authorization. By including more data, such as the app's privacy policy and bytecode, TAPVerifier by Yu et al. [27] expands the consistency verification. Using code features and methods from natural language processing, AutoPPG by Yu et al. [28], developed by the same research team, infers the privacy policies of apps. AsDroid by Huang et al. [29] uses mismatches between user interfaces and underlying code properties to detect stealthy behaviours in code. Wijesekera et al. [30] employ context for classification models that is probably accessible to the user and customize the models; they may be the first to automatically infer privacy decisions on a case-by-case basis at runtime without direct user involvement. Nguyen et al. [31] assess user perception about Android app behaviour based on GUI features to identify access to sensitive resources. In order to do static analysis and identify violations of the use of sensitive information, Zhang et al. [32] developed a language for formalizing GUI regulations and presented an abstraction called *event-driven layout forest*. A study of user behaviour regarding the granting and denying of Android app permissions is undertaken by Cao et al. [33]. Our approach differs from those studies in that we utilize graph embedding techniques to encode app behaviours represented by inter-procedural control-flow graphs and build embedding-based classification models for permission (mis)uses.

D. Deep Learning-Based Vulnerability/Malware Detection

Deep Learning based techniques are also used in Android app analysis and malware detection. Malviya and Gupta apply LSTM on Android apps opcodes to detect malicious behaviour of Android apps [34]. De-LADY by Sihag et al. [35] is an obfuscation-resilient approach which analyses the

TABLE I
SUMMARY OF RELATED WORKS

Research Area	Granularity	Analysis Method	Data Used
Understanding and Managing Privacy [2], [8], [9]	Appwise	Dynamic	Network traffic, Contextual information
Detection of Abnormal Uses of APIs, Permissions, and Private Data [10], [11], [12], [13], [14], [16], [17], [18]	Appwise	Static besides Jamrozik et al. [13] and Zhang et al. [16]	App descriptions, Data flow, GUI mining, App code, UI and context information
User Perception versus App Behaviour [19], [20], [21], [23], [24], [25], [27], [28], [29], [29], [30], [31], [32], [33]	Appwise except Wijesekera et al. [30] which is functionality wise	Static except Yang et al. [21]	Permission event graph, App functionalities, GUI change event sequence, Semantic annotations for UI elements, Event surrounding private data usage, GUI features and App's description, privacy policies, and bytecodes
Deep Learning-Based Vulnerability/Malware Detection [34], [35], [36], [37]	Appwise	Static except Sihag et al. [35]	Dynamic analysis logs, API calls, permissions
Permission Classification/Prediction [38], [39], [40], [41], [42]	Appwise	Static except Shaozhang et al. [40]	Apps level of protection and threat, collaborative filtering, text mining, Categories, permissions and textual description of app, users' privacy decisions

logs generated by dynamic analysis of the Android app and extracts behavioural patterns for training deep learning models. Elayan and Mustafa [36] use a specialized Recurrent Neural Network (RNN) model named Gated Recurrent Unit (GRU) for detecting malware using API calls and Permissions.

Our approach uses graph embedding for permission classification. Similar deep graph learning techniques have been used for vulnerability detection, malware detection, code classification, third-party code detection, and other tasks [43]. For example, Kim et al. [37] and DeepFlow [44] use various kinds of representations of Android apps and deep learning techniques to detect malware. VulDeePecker [45] and BGNN4VD [46] use combinations of syntax structures and control-/data-flow graphs to represent programs and employ various deep neural network models to learn vulnerability patterns for C/C++ programs. However, as far as we know, no study has been done for fine-grained Android permission classification per functionality using inter-procedural control-flow graphs.

E. Permission Classification/Prediction

Ashawa and Morris [38] categorize malware permission requests using the level of protection and threat in Android apps. Bao et al. [39] propose an Android permission recommendation system with two different strategies, i.e., collaborative filtering and text mining. Shaozhang et al. [40] propose a machine learning-based dynamic permission management system for Android apps using a dynamic permission management database. Approach by Deguang and Hongxia [41] makes use of a structural feature learning framework to pinpoint the connections between app categories, permissions, and textual descriptions and forecasts permissions. Mendes et al. [42] propose an automated and tailored method to predict user preferences for permission requests after analyzing users' privacy decisions and the situation.

To the best of our knowledge, except for some approaches based on dynamic analysis [30], all existing classification and permission studies consider an app as a whole and have not considered individual functionality associated with an event handler linked to a user action.

F. Differences from Related Work

After the review of related works as summarised in Table I, we note the following common shortcomings in those studies:

- Except for a small number of studies (like [30]), other works on app-level or library/package-level are not fine-grained or flexible enough for different contexts and app usages;
- Relationship between app functionality and its permission uses is often considered in the context of the same app or the apps of the same categories; similarities and differences in the relations *across different apps* have not been utilized for classification and legitimacy decisions.

Our approach aims to overcome these shortcomings via:

- More fine-grained detection of permission misuses in the context of each functionality linked to user actions, instead of whole application-level malware detection.
- More effective encoding of app behaviours and permission use that automatically and holistically considers similarities and differences across various apps to facilitate accurate legitimacy decision-making.

Our approach is more fine-grained, which means that we can identify specific malicious behaviours that other approaches might not be able to detect. Because of this, our approach identified some misuse of permissions even in apps with good reputations as shown in our evaluation (Section IV), which would not be detected as malware by other approaches.

III. SOLUTION DESIGN AND IMPLEMENTATION

This section first presents the overall idea and design of our approach in III-A, and then describes more details for various components in our approach in III-B—III-E.

A. The Proposed Approach

As introduced in Section I, our approach aims to improve the mechanism for supporting automated permission classification and privacy protection for users. In particular, we aim to make the identification of permission misuses in Android apps more fine-grained in the context of event handlers related to user actions.

The main technical objective of our approach is to classify and predict the permission(s) necessary for each function triggered by a user in an Android app (e.g., via simply launching the app or tapping a widget on the graphical user interface of the app). Then, the differences between the predicted permission use and the actual permission use by the app can be used to warn the users about potential misuse.

In brief, our approach integrates static analysis of app bytecode associated with event handlers of user interface and Android system events with machine learning techniques to infer patterns of permission (mis)uses across a variety of apps. It is put into practice in four components.

- App's static analysis is the first component. It creates inter-procedural control flow graphs (ICFGs) that are connected to event handlers (e.g., triggered by a user action via the app user interface or a system event that may not be visible to the user).
- Embedding of the inter-procedural graphs is the second component. It creates vectors from the graphs that are utilized in classifying and predicting permissions.
- Utilizing user impressions of app functionality and permissions is the third component (UI events and handlers). These perceptions are shaped by user studies conducted on actual apps and assist the supervised classifier in predicting permissions.
- A supervised learning classifier is the fourth component. This classifier predicts permissions needed by the app functionality represented as the ICFG embeddings, given the previous three components, and identifies permission misuses by utilizing different but related perspectives on the app's functionality and permission needs.

Figure 2 shows an overview of our approach. The approach has two phases. The first phase is the training of a supervised classifier and the second phase is to apply this trained classifier for predicting permissions. In our approach, the bytecode of an Android app initially undergoes static code analysis, as illustrated in the figure. There are two steps to performing static analysis. To begin, FlowDroid [47] based on the Soot framework [48] is used to create a call graph combined with a control flow graph to produce directional graphs (digraphs). In the second step, we analyse digraphs and assign labels in the form of permissions using the customized Androwarn [49] tool. The baseline labels that the customized Androwarn automatically generates are expanded and validated manually. The inter-procedural control flow graphs produced by FlowDroid are used in this manual verification. In order to generate the embedding vectors, we conduct whole graph embedding on the inter-procedural control-flow graphs of Android apps that we obtained from FlowDroid. The supervised learning classifier is now fed the confirmed labels obtained after manual verification and the embedding vectors produced by graph embedding for evaluation and building a model for predicting permissions.

In the application phase, an Android app goes through static analysis with Soot which generates inter-procedural control flow graphs for various event handlers in the app. Embedding vectors are then generated from the graphs using graph embedding. These vectors are then fed to the trained classifier model to predict permission. More information about these actions is provided in the following subsections.

B. Static Program Analysis

Static code analysis is a technique for looking over codes before a program is run. A widely used tool for static analysis of Android apps is the Soot framework [48]. To load Android apps and produce call graphs along with control flow graphs, Soot and tools built from it, such as FlowDroid [47] and Gator [50], are used in the first step of static analysis. These graphs serve as the fundamental building blocks for graph embedding. Figure 3 displays an example of a call graph along with a control flow graph generated from an app. The nodes in the graph contain the instructions from the app's code in the format of Soot's Jimple intermediate representation.

We first build digraphs from call graphs utilizing the Class Hierarchy Analysis (CHA) and Soot Pointer Analysis Research Kit (SPARK) techniques. We extract features from discovered GUIs, API signatures, digraphs, and permissions used in the app by integrating the call graph findings from both SPARK and CHA algorithms. Android lifecycle methods (such as onCreate, onStart) and event methods (such as onClick, onItemClick) are graphed, with API signatures in digraph nodes and used Android permissions (obtained from jellybean and axplorer mappings that map API calls to the permissions used [51]) maintained in digraph names. We improve the digraphs by creating a control flow graph for the method body and assigning labels to the digraph edges. The graph representation is enhanced by combining control flow graphs and edge labels with the call graphs to incorporate more information (as illustrated in Figure 3). The output of the first step of static analysis is a set of inter-procedural control flow graphs (ICFGs), one for each event handler. The graphs are stored as directional graphs (digraphs) in the DOT file format [52] and are fed to the next step for graph embedding.

We use another tool Androwarn [49] in the static analysis component. Androwarn is a malicious behaviour detection tool for Android apps. It statically analyses the Android app's Dalvik bytecode for potentially malicious behaviour. We customized the Androwarn to work on digraphs for generating automated labels in the form of permissions, for the purpose of generating training data in our work (see more details in Section III-D1 and III-E).

C. Graph Embedding

This component of our approach is responsible to identify permission (mis)use patterns in inter-procedural control-flow graphs (ICFGs) relevant to machine learning techniques. A graph is made up of a number of nodes (vertices) and a number of edges connecting these nodes. Nodes in graphs are not randomly scattered and independent because they are invariably connected through edges, forming potentially complex patterns. Traditional machine learning techniques are not easily able to handle computational issues on graphs. There are two strategies to apply machine learning to graphs. One

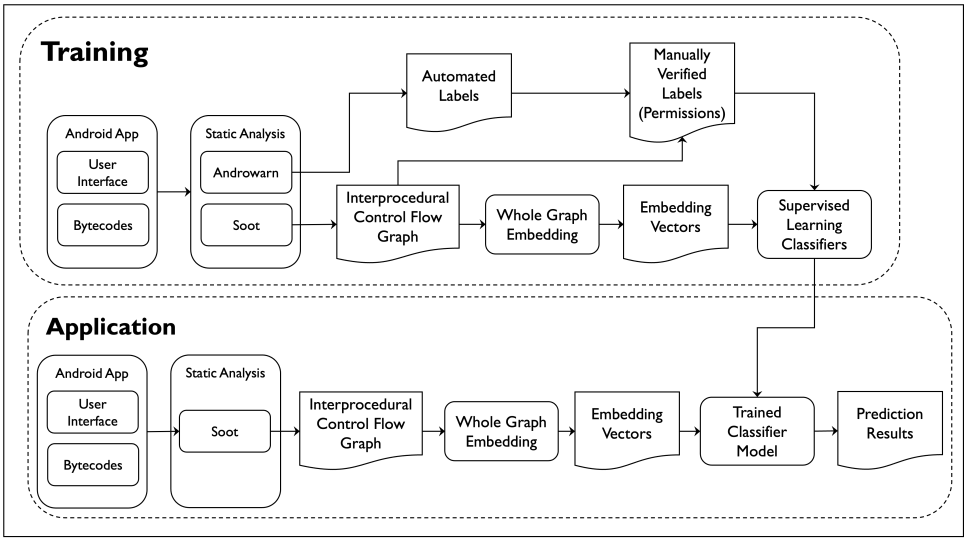

Fig. 2. Our Approach based on Static Analysis, Graph Embedding, User Study and Labelling and Supervised Classification

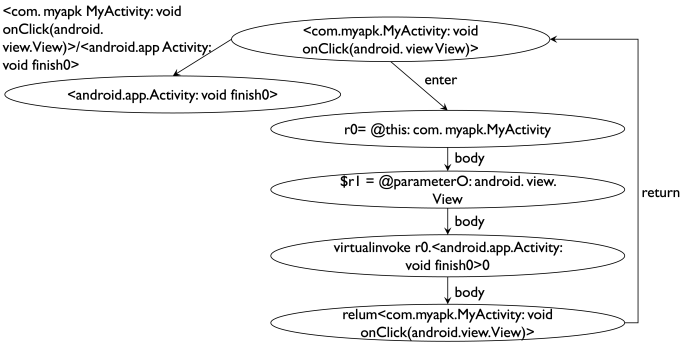

Fig. 3. Example Call Graph combined with Control Flow Graph

strategy is to create a brand-new classification mechanism for graphs. This new classification mechanism is known as collective classification [53]. Collective classification considers mapping for a node's neighbourhood and the mapping between its attributes and label. The other strategy is to flatten a graph by building a set of features to identify each node. In this case, traditional classification techniques can be applied after the graphs are flattened, therefore it has become more and more well-established [54].

Graph embedding belongs to the second strategy. It converts graphs to numerical vectors in high-dimension spaces while maintaining their attributes [55], so-called embedding vectors. In order to detect desired properties of the graphs, machine learning models are trained using these graph embedding vectors. In this work, we perform graph embedding on inter-procedural control flow graphs of Android apps obtained from static analysis with Soot. We use NetworkX [56] library for loading graphs in the DOT format and GL2Vec [57] in the karateclub machine learning library [58] for obtaining embedding vectors from inter-procedural control flow graph. NetworkX is a network analysis package and it provides an implementation of different graph algorithms and graph data structures including digraph which is being analysed in this work. GL2Vec provides a whole graph embedding implementation of the creation of a line graph of each graph and Weisfeiler-Lehman tree features for nodes with degree centrality. It is an improved version of Graph2Vec [59]. Since it is able to handle edge labels and structural information, we selected it for our implementation. We used the default implementation of GL2Vec with the following parameters: wl_iterations=2, dimensions=128, workers=4, down_sampling=0.0001, epochs=1000, learning_rate=0.025, min_count=5, seed=42.

D. User Study and Graph Labeling

Our approach uses supervised machine learning techniques to differentiate the code graph patterns of benign permission uses from those of malicious uses for identifying permission (mis)uses. To train such machine learning classifiers, we need to provide sample data together with known benign or malicious behaviours, i.e., inter-procedural control-flow graphs associated with event handlers that either contain no permission uses or contain known benign or malicious uses in the context of our paper. In our context, the labels for the training data have two types:

- Label 1 for an ICFG containing malicious misuses of permissions;
- Label 0 for an ICFG that either does not use permissions or uses permission benignly in relation to the functionality related to the event handlers of the user interfaces that may be visible to users.

We performed graph labelling in two steps, first by customized Androwarn during static analysis and malicious behaviour detection, and then manually verified by ourselves. The next two subsections describe both steps in more detail.

1) Automated Labeling: Due to a huge number of inter-procedural control-flow graphs (ICFGs), we customized Androwarn [49] to help scale down the manual effort needed to label the graphs. We manually discovered a collection of

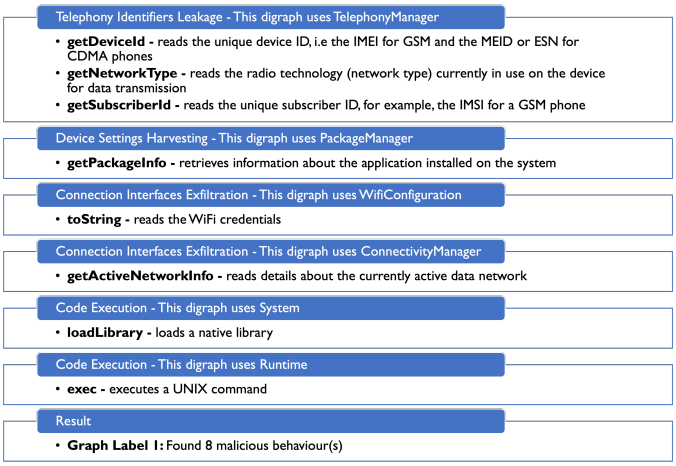

Fig. 4. An example output of a malicious graph (Label 1) by Androwarn

definitions of potential malicious behaviours in Androwarn's source codes. Malicious behaviour as detected by Androwarn is indicated by specific uses of Android SDK classes and API method names (such as TelephonyManager and getDeviceId) by keyword matching only. Because the use of these classes and API methods require permissions related to the potential private information leakage, Androwarn would identify all such uses without considering whether the permission uses are necessary for the app's functionality. The categories detected by Androwarn are as follows: (1) Telephony Identifiers Leakage (2) Device Settings Harvesting (3) Location Lookup (4) Connection Interfaces Exfiltration (5) Telephony Services Abuse (6) Audio Video Eavesdropping (7) Suspicious Connection Establishment (8) Pim Data Leakage (9) Code Execution.

In its default implementation, Androwarn searches for such classes and API keywords in the *whole* app package to detect if an app as a whole may be malicious. We modified Androwarn's source codes so that it may detect the uses of such classes and APIs at the *graph* level to generate the baseline graph labels. Because of this change, Androwarn instead of searching the whole app package, searches at the granularity of ICFGs, using the same class and API keywords.

An ICFG is assigned a Label 0 when there is no malicious behaviour detected or a Label 1 when there are malicious behaviours found based on our customized Androwarn search. Figure 4 shows an example output of Label 1 for an ICFG. The figure shows that the functionality associated with the graph contains potential misuse by 8 APIs that need permissions.

We also included training data from known malicious Android apps. We could presume that all code functionalities and permission use in malware were erroneous, but we nevertheless think some fine-grained code in malware may perform benign functionality in order to disguise itself, and treating all ICFGs from malware as malicious may hurt the accuracy of the training data, the training data, especially for those with Label 0s. Thus, when we ran our customized Androwarn on individual ICFGs of malware, we only took the ones with reported malicious behaviours, and labelled them with 1 for training data, but removed those cases without reported malicious behaviours (i.e., do not use them either for Label 0 or 1 training data) to avoid introducing noises into the training data. To further improve the labels for the code graphs and enhance the quality of the subsequent machine learning models, we further make use of manual analysis of the graphs which is explained in the next subsection.

2) Manual Labeling: Labels obtained from our customized Androwarn, especially the Label 1s for non-malware apps, are then verified manually in this step. This whole process takes place with the help of inter-procedural control flow graphs obtained from FlowDroid in *DOT* format. In order to explore the extent of malicious behaviours reported by customized Androwarn, the ICFGs related to these behaviours are searched to discover nodes consisting of malicious behaviours. All nodes and edges with data and/or control dependencies are then consolidated to create subgraphs. Visualization of the subgraphs is then further generated to facilitate manual analysis of the functionality of the ICFGs.

Following the principle of user awareness for privacy protection, we believe that there should be some user interface elements in the app to notify the user if permission involving private data is used in its code; if there are no such UI elements for the user, which means the app code for the ICFG uses the (granted) permission without informing the user, it should be considered as potentially malicious. Thus, during the manual analysis of ICFGs, we also look for the presence of classes and/or event methods that suggest a user interface (UI). Further analysis (e.g., running the app, reading its XML layout files) may be required if the existing knowledge contradicts the results of static analysis of Androwarn or if the analysis does not provide a full picture of the UI.

For example, Figure 5 depicts part of a sample graph, extracted from an ICFG for an app. At the very bottom, the node showcases the API method `getDeviceId()` that uses privacy-sensitive permission `READ_PRIVILEGED_PHONE_STATE`. Following the edges and nodes in the ICFG leading to the permission, the method `setAnimationListener()` together with the class Animation suggests that there is a UI present. The purpose of the interface can also be further inferred to be a splash advertisement when we continue to follow the control flow. With this information, it can be understood that the UI is for displaying ads and does not accurately reflect the permission of the user and is thus labelled as 1 (malicious) in this regard.

For another example, the part of a graph shown in Figure 6 has its map view evidently needing the method `getActiveNetworkInfo()` requiring the permission `ACCESS_NETWORK_STATE`, which provides details of the device's current network, in order to connect to the internet. More crucially, it is quite common for an average user to understand that permission is needed to connect to the internet when looking at the app's interface. So, we label this case as 0 (benign) even though Androwarn suggests it may be malicious.

For each ICFG labelled as 1 by our customized Androwarn, we thus manually analyze all the permission uses in the ICFG

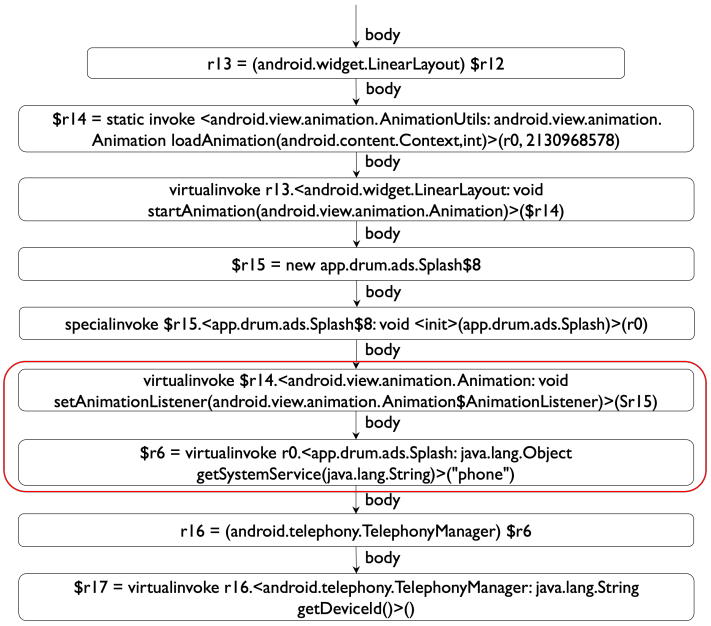

Fig. 5. Digraph depicting Malicious Behaviour

individually on whether they are reflected through the UI, and update the graph's label to 0 if all of the permissions are determined to be benign. Section IV-A gives more details on the apps that we have curated labels.

E. Supervised Learning Classifier

With our curated benign/malicious labels for inter-procedural control-flow graphs and their embeddings as input, we can now train machine learning classifiers to recognize permission (mis)uses using the embedding vectors as features for classification. We note that an ensemble classification using multiple classifiers usually produces better results than a typical classification model using a single classifier. We use sklearn library [60] for Ensemble Graph Classification. To balance the shortcomings of individual classifiers and enhance the quality of predictions, we developed a VotingClassifier based on majority voting from the following seven widely used machine learning classifiers: k-Nearest Neighbors, Support Vector Machine, Decision Tree, Random Forest, AdaBoost, Gradient Boosting and eXtreme Gradient Boosting. The trained classifier model can then be applied to any new given app to make predictions about its permission (mis)uses. We used default parameters for the classifiers from the library. The details of their parameters can be found in our GitHub repo.

IV. Evaluation

In this work, we explored fine-grained context-sensitive permission usage analysis and thereby identify misuses. For this, we specifically addressed the following research questions:

- **RQ1:** How effective is the approach in identifying the (mis)use of permissions and users' private data?
- **RQ2:** What are the reasons behind the (mis)use of permissions and users' private data?
- **RQ3:** What are factors which can affect the performance of the approach?

This section explains the evaluation process for our work and answers the research questions. Details of the evaluation, the dataset used, evaluation metrics, and results of graph labelling and supervised classification are described here.

TABLE II
Graph Labeling Results

Apps	#Apps	Auto 0	Auto 1	Manual 0	Manual 1	Total
SG Smart Nation Apps	19	544	166	625	85	710
Androzoo Apps	30	737	277	889	125	1014
DeepIntent Malicious Apps	16	-	289	-	289	289
More Malicious Apps	24	-	135	-	135	135
Total	**89**	**1281**	**867**	**1514**	**634**	**2148**

A. Dataset

We selected apps from different sources and varieties to have a diverse set for our experiments. Our evaluation dataset consists of 89 apps. 49 apps of them are benign and 40 apps are malicious. 19 benign apps are Singapore smart nation apps [61] with a good reputation, and 30 apps are from Androzoo dataset [62]. We selected Singapore smart nation apps due to our familiarity with the apps as these apps are used quite frequently in Singapore and due to which manual labelling became easier. We selected the apps from the Androzoo dataset by checking the functionality of the apps so that they may be different from the SG smart nation apps. Out of 40 malicious apps, 16 are obtained from DeepIntent [63] and 24 are collected from various other resources. These apps are also selected with the aim of making a dataset of diverse apps. We first selected apps from DeepIntent and then collected apps from other sources by checking their similarity with DeepIntent apps. A list of all apps used in this work is available at https://github.com/ervikas/DroidGem.

These apps first go through our graph labelling process (Section III-D). Table II shows the results of graph labelling. Auto 0 and Auto 1 represent the labelling done by our customized Androwarn, and Manual 0 and Manual 1 represent manual labelling. From the 19 SG smart nation apps 544 benign and 166 malicious graph labels are given in automated labelling, while 625 benign and 85 malicious graph labels are given after manual labelling. Similarly from Androzoo apps 737 benign and 277 malicious labels are given in automated labelling, while 889 benign and 125 malicious labels are given after manual labelling. From DeepIntent malicious apps 289 malicious labels are given. Similarly, in the case of other malicious apps 135 malicious labels are given. We got a total of 2148 labelled graphs for training and testing. As noted in Section III-D1 and Section III-D2, we do not use Auto 0 labels from malicious apps to avoid introducing noises into Label 0 training data, and we assume all Auto 1 labels from malicious apps are indeed malicious to reduce manual labelling efforts.

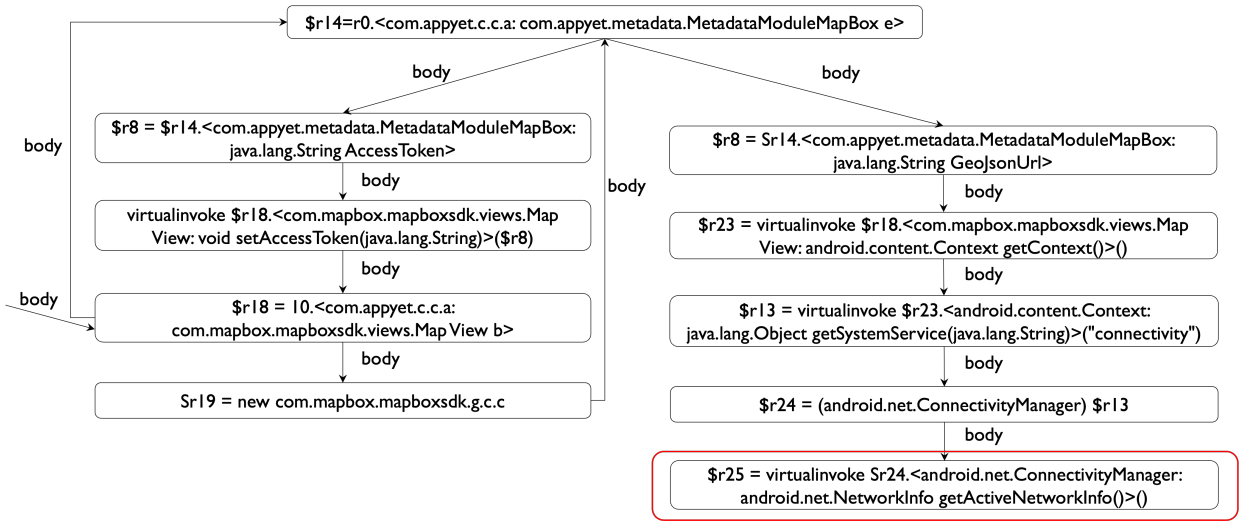

Fig. 6. Digraph depicting Benign Behaviour

The dataset of the graphs and their labels are then split on the commonly used 80/20 ratio for later machine learning, with 80% for training and 20% for testing.

B. Performance Metrics

The performance of a machine learning classifier is often evaluated in terms of precision, recall, and F1 score. In this work, a graph is deemed legitimate if no malicious behaviour is observed (with a label of 0) and illegitimate if any malicious behaviour is discovered (with a label of 1). The performance metrics are explained in the following sections.

1) Precision: Precision measures the correct predictions made out of total true positive predictions. The formula for precision is: $Precision = \frac{TruePositives}{TruePositives+FalsePositives}$.

We calculate our performance metrics in two different cases based on whether Label 0 or Label 1 is considered positive. First, we calculate the metrics considering benign as positive. Second, malicious is considered positive for the calculation.

2) Recall: Recall measures the correctly predicted positive cases, over all the positive cases. It is also called Sensitivity. The formula for the recall is: $Recall = \frac{TruePositives}{TruePositives+FalseNegatives}$.

3) F1 Score: F1-Score combines both precision and recall. It is the harmonic mean of precision and recall. The formula used to calculate F1-score is: $F1Score = 2 * \frac{Precision*Recall}{Precision+Recall}$.

C. Results and Discussion

1) Permission Prediction Results: The results of the evaluation provide an answer to the first research question (RQ1). Table III and IV shows the performance of our supervised classification model applied to the 20% test data after the model is trained via a 10-fold cross-validation using the manually verified labels. The confusion matrix in Table III tells the numbers of graphs that our model makes correct or wrong predictions: 307 Label-0 predictions and 102 Label-1 predictions are correct, while 4 Label-0 and 17 Label-1 predictions are wrong. We then calculate precisions, recalls, and F1 scores based on the confusion matrix according to the formula in Section IV-B. As shown in Table IV, our ensemble model resulted in a promising precision and recall of up to 96% and 86% respectively for detecting code graphs that may contain malicious or user-imperceptible permission uses.

TABLE III
CONFUSION MATRIX OF ENSEMBLE CLASSIFIER ON TEST DATA

		Predicted 0	1
Actual	0	307	4
	1	17	102

TABLE IV
ENSEMBLE CLASSIFICATION REPORT (USER MANUAL LABEL)

	Precision	**Recall**	**F1 Score**
Label 0 (benign)	0.95	0.99	0.97
Label 1 (malicious)	0.96	0.86	0.91
Weighted Avg	0.95	0.95	0.95

2) Permission Misuse Analysis: We did this analysis to find an answer to the second research question (RQ2). As shown in Tables II and III, DROIDGEM correctly identifies many ICFGs in the benign apps that misuse permissions along with manual verification. We look into these cases and manually categorize the reasons why the apps (mis)use the permissions.

The majority of the DOT files identified as malicious are found to have analytic and marketing aims. The developers' desire to gather as much information from their users as possible is perhaps one of the reasons. A large amount of data aids developers to understand their users' demographics, preferences, and usage habits. Another factor that could contribute to developers using permissions needlessly is that users may not comprehend the reasons for the permissions being used and may have limited options if they want to utilize the applications. Applications that generate revenue from advertisements have a greater incentive to show ads to the target audience that the advertiser has specified. The applications we studied also frequently had more and more opportunities for data analysis, which is a major motivation for increased data collecting.

Some developers are also highly dependent on available third-party libraries and tools. Regardless of the degree of openness, developers sometimes have little control and/or knowledge of the libraries' usage of permissions. Furthermore, much like the application's users, developers may be requested by the libraries and tools' owners to use certain permissions in order for integration. These libraries and tools are also often developed with them being general purposes in mind, hence unnecessary permissions may be included.

3) Prediction Error Analysis: This analysis is intended to answer the third research question (RQ3). Our manual labelling brings about human knowledge to ensure labels can be properly justified for training better machine learning classifiers. However, our prediction results show that there are still wrong predictions for about 5% (21 out of 430) of test code graphs from various apps used for testing.

Even if permissions are designated independently depending on whether they utilize UI elements, the ICFGs would be categorized as harmful as long as one permission is labelled as such. As a result, the algorithm may identify additional occurrences of the permissions as malicious as well, even though the harmful label does not apply to each occurrence of the permissions individually. One such instance is ICFG of an Androzoo program that is flagged as dangerous, although only 2 of the application's 6 permissions are devoid of a UI element. Then, it is shown that this specific ICFG was one of the incorrect predictions.

Human-related factors may also have a significant role in the production of false predictions. In addition to the apparent human faults at work (wrong labels), the labels will also be impacted by the labeller's comprehension of the various APKs, methods, classes, and libraries. Based on a comparison between the ICFGs that have been wrongly labelled by the algorithm and annotations after manual labelling, it is suggested that prior knowledge of the applications may change, which might result in different approaches to manually labelling the DOT files. Similarly, the labeller might have a better insight into certain applications, for example, SG Smart Nation essential applications, leading to biases.

Another human-related reason would be that certain digraphs are found difficult to be generated and read because of the encryption applied to apps before releasing them. As a result, manual labelling may be more difficult and produce outcomes with fewer justifications since it is more difficult to establish an overview. This is particularly relevant to Singapore Smart Nation applications because those apps use more stringent encryption.

Lastly, the benefit of manual labelling may hinder a more repeatable approach to labelling permissions. The ICFG-generated sub-graphs provide information about a particular questionable permissions' intended use. Nearly all of the handful of Androzoo applications whose outcomes were wrongly predicted have been found to be tied to analytics or advertising. Despite having a UI element overall, excessive usage of rights that are not particularly related to their intended function has resulted in certain ICFGs being classified as harmful while other ICFGs with comparable permissions that are believed to serve other goals are not.

A possible future solution to these above-mentioned issues is to augment DROIDGEM with dynamic analysis. Dynamic analysis can create a real-time mapping between event handlers and permissions usage which will help to verify the generated/manually verified labels.

D. Threats to Validity

1) Threats to Construct Validity: A major threat is whether the graph embedding of inter-procedural control-flow graphs (ICFGs) is effective in capturing the functionality of an app. Based on related work on using various code representations and deep learning for malware detection and classification (Section II-D) and our empirical evaluation, such techniques can indeed capture much of the code functionality, although it needs further research to investigate to what degree such techniques capture code semantics.

2) Threats to External Validity: Major threats are sample bias and evaluation bias. In our evaluation, we used a diverse dataset from different sources to improve app variations and scenarios. We also included benign apps that are not supposed to contain permission misuses for evaluation. And we used commonly used metrics such as precision, recall, and F1-score for measuring the prediction performance. We also employed 10-fold cross-validation to reduce the effect of randomness.

3) Threats to Internal Validity: Our code implementation, human labelling process, and static analysis tools used may all contain limitations and errors. Internal validity is also at risk from potential discrepancies and randomness throughout various runs. Multiple authors of the paper went through our code and dataset and performed multiple checks and achieved stable prediction accuracy to reduce such threats.

E. Limitations and Possible Improvements

Our approach and the evaluation show that fine-grained context-aware prediction of permission (mis)uses can be achieved. On the other hand, we note that there are various kinds of limitations in our current design, implementation, and evaluation, which can be further improved.

1) On the Design of the Approach: According to some studies in the literature, it may be possible to predict malicious intentions of an Android app by just checking the dangerous permissions used in the app (The Android system has a list of dangerous permissions [64]); i.e., if an app uses more dangerous permissions, it is more likely to exhibit malicious behaviour. We ran an analysis to see how effective this alternate notion is. First, we calculated the number of dangerous permissions being used by the malicious and benign apps from our dataset. We calculated the average and standard deviation of the number of dangerous permissions used in the apps (shown in Table V). As shown in the table, malicious apps on average use about one more dangerous permissions than benign apps. But the difference is not significant enough, considering that their standard deviations are higher than

TABLE V
ANALYSIS OF DANGEROUS PERMISSIONS USED IN APPS

	Number of Apps	Average	Standard Deviation
Benign	49	3.27	2.89
Malicious	40	4.36	2.35

two. This shows that using dangerous permissions alone as a criterion for predicting the intentions of an app is insufficient.

DROIDGEM uses custom-built inter-procedural control-flow graphs to represent the functionality and behaviour of UI-linked event handlers, which may not be optimal. In the literature, various kinds of code representations, such as program dependency graphs [65], code property graphs [66], permission event graph [19], etc., have been utilized to classify code functionality [66] and detect malware [65]. Such graph representations can potentially be incorporated into DROIDGEM to help identify functionality more accurately. Also, GUI contexts, such as UI widget types, icon images, layouts, transitions to next windows, etc. [17], [63], may also be used to improve prediction accuracy.

DROIDGEM analyses only static details of an app without using dynamic information. We intend to augment DROIDGEM with information from dynamic analysis to overcome this limitation. We also see the usage of unsupervised learning algorithms for detecting abnormal permission uses, which would lessen the requirement for manually labelled user perceptions.

2) On the Labelled Permission Usage Data for Training: For malicious apps, we used automated analysis and assumed all permission uses in malicious apps as illegitimate. Since malicious apps are dangerous to the privacy and security of users, they should not be run by normal users at all, even if some parts of the malicious apps are harmless. Also, we try to capture as many potential misuses as possible at the cost of treating some harmless uses as illegitimate. We might use malware detection methods in our approach for an app first, and then if the program is benign would we use our fine-grained permission detection. We can reduce false positives in this way. Additionally, in the future, when we gather more labelled data, we might be able to refine/retrain the models using training data from benign apps solely.

3) On the Implementation: DROIDGEM uses Soot, FlowDroid and other tools built on it along with Androwarn for the static analysis of Android apps. All these tools work on native Android apps. These tools have very limited or no support for non-native apps. This limits the usability of DROIDGEM in analysing non-native apps. Lee and Wu [67] analyzed cross-platform or HTML-based hybrid apps and suggested directions for the analysis. We want to use their suggestions and additional information, such as package structures, app GUI layouts, and execution profiles, for more comprehensive and versatile analyses in our future work.

The tools used in DROIDGEM have limited or no support for obfuscated apps. This limits the performance of DROIDGEM. The solution is to use de-obfuscation techniques and dynamic analyses. It will not overcome the problem completely but the effectiveness of DROIDGEM can be improved.

Another shortcoming of DROIDGEM is the lack of sufficient labels. We intend to increase the size of the dataset to overcome this limitation.

4) On the Evaluation and More Applicable Scenarios: Although DROIDGEM performed well in evaluation, there is the possibility of improvement as the size of the app dataset is not large. So generalization of results may not be effective. Our intention is to increase the number of apps for experiments.

Our approach is a static analysis-based approach which is suitable for *offline* analysis of Android apps. It may be used by an app store to check whether an app complies with more rigorous privacy-awareness policies before publishing an app. It may also be used by a user to check an app before using it. It might be utilized as a preliminary step before performing a deeper analysis of an app; further investigation can validate the labels that our approach generates, which may be a sign of malicious behaviour. It will reduce the amount of time required to conduct an in-depth analysis of the app. Our methodology can assist application stores in establishing rules for the handling of personal data and determining whether the features of an app available there comply with those requirements, and help to improve privacy awareness among both app developers and users.

V. CONCLUSION AND FUTURE WORK

This paper presents an approach for fine-grained Android app permission classification using graph embedding. Graph embedding is performed on the inter-procedural control-flow graphs representing functionalities of an app that may be triggered by users' or systems' events. Feature vectors obtained from graph embedding are evaluated with supervised ensemble classifiers to identify the permission (mis)uses. We have implemented our approach as a prototype, named DROIDGEM, and evaluated it on 89 apps, out of which 49 apps are benign and 40 apps are malicious. The results are promising since, compared to manually validated permission uses, our approach can classify whether permissions utilized by an app's functionalities are legitimate with up to 95% precision and recall.

We are aware of a number of limitations in our approach, but we are certain that these can be overcome with better app UI and code analysis and learning techniques in further work. By testing on larger, more recent releases of Android app sets, we are committed to enhancing DROIDGEM and its results.

We have published data and source code of our implementation and evaluation at https://github.com/ervikas/DroidGem.

ACKNOWLEDGMENT

This research is supported by the National Research Foundation, Singapore, and the Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme, National Satellite of Excellence in Mobile Systems Security and Cloud Security (NRF2018NCR-NSOE004-0001). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore and the Cyber Security Agency of Singapore.

REFERENCES

[1] "Mobile OS worldwide market share," https://gs.statcounter.com/os-market-share/mobile/worldwide, accessed on: 03-05-2023.

[2] J. Hong, "The privacy landscape of pervasive computing," *IEEE Pervasive Computing*, vol. 16, no. 3, pp. 40–48, 2017.

[3] J. Constine. (2019) Facebook pays teens to install vpn that spies on them. [Online]. Available: https://techcrunch.com/2019/01/29/facebook-project-atlas/

[4] M. Shealy. (2019) What facebook's vpn scandal tells us about its approach to data privacy. [Online]. Available: https://channels.theinnovationenterprise.com/articles/facebook-pays-teens-to-download-a-vpn-that-spy-s-on-their-activity

[5] W. Zack, C. Josh, and L. Ingrid. (2019) Google will stop peddling a data collector through apple's back door. [Online]. Available: https://techcrunch.com/2019/01/30/googles-also-peddling-a-data-collector-through-apples-back-door/

[6] F. H. Cate, "The limits of notice and choice," *IEEE Security & Privacy*, vol. 8, no. 2, pp. 59–62, 2010.

[7] L. F. Cranor, "Necessary but not sufficient: Standardized mechanisms for privacy notice and choice," *J. on Telecomm. & High Tech. L.*, vol. 10, p. 273, 2012.

[8] H. Jin, M. Liu, K. Dodhia, Y. Li, G. K. Srivastava, M. Fredrikson, Y. Agarwal, and J. I. Hong, "Why are they collecting my data?: Inferring the purposes of network traffic in mobile apps," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 2, pp. 1–27, 12 2018.

[9] S. Chitkara, N. Gothoskar, S. Harish, J. I. Hong, and Y. Agarwal, "Does this app really need my location?: Context-aware privacy management for smartphones," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 1, pp. 1–22, 09 2017.

[10] Y. Li, F. Chen, T. J.-J. Li, Y. Guo, G. Huang, M. Fredrikson, Y. Agarwal, and J. I. Hong, "Privacystreams: Enabling transparency in personal data processing for mobile apps," *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, vol. 1, pp. 76:1—-76:26, 09 2017.

[11] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *ICSE '14: Proceedings of the 2014 International Conference on Software Engineering*. ACM Press, Jun. 2014, pp. 292–302.

[12] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 426–436.

[13] K. Jamrozik, P. von Styp-Rekowsky, and A. Zeller, "Mining sandboxes," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 37–48.

[14] V. Avdiienko, K. Kuznetsov, I. Rommelfanger, A. Rau, A. Gorla, and A. Zeller, "Detecting behavior anomalies in graphical user interfaces," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 201–203.

[15] J. Hotzkow, "Automatically inferring and enforcing user expectations," *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017.

[16] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in android apps with permission use analysis," *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013.

[17] H. Fu, Z. Zheng, S. Zhu, and P. Mohapatra, "Keeping context in mind: Automating mobile app access control with user interface inspection," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 2089–2097.

[18] K. Olejnik, I. Dacosta, J. S. Machado, K. Huguenin, M. E. Khan, and J.-P. Hubaux, "Smarper: Context-aware and automatic runtime-permissions for mobile devices," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 1058–1076.

[19] K. Z. Chen, N. M. Johnson, V. V. D'Silva, S. Dai, K. MacNamara, T. R. Magrino, E. X. Wu, M. C. Rinard, and D. X. Song, "Contextual policy enforcement in android applications with permission event graphs," in *NDSS*, 2013.

[20] X. Chen and S. Zhu, "Droidjust: automated functionality-aware privacy leakage analysis for android applications," *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2015.

[21] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "Appintent: analyzing sensitive data transmission in android for privacy leakage detection," *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013.

[22] H. Fu, Z. Zheng, A. K. Das, P. H. Pathak, P. Hu, and P. Mohapatra, "Flowintent: Detecting privacy leakage from user intention to network traffic mapping," in *2016 13th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, 2016, pp. 1–9.

[23] E. Fernandes, O. Riva, and S. Nath, "Appstract: On-the-fly app content semantics with better privacy," in *ACM Annual International Conference on Mobile Computing and Networking (MobiCom'16)*. ACM, July 2016. [Online]. Available: https://www.microsoft.com/en-us/research/publication/appstract-on-the-fly-app-content-semantics-with-better-privacy/

[24] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "Appcontext: Differentiating malicious and benign mobile app behaviors using context," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 303–313.

[25] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "WHYPER: Towards automating risk assessment of mobile applications," in *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX Association, Aug. 2013, pp. 527–542. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/pandita

[26] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in android applications," *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[27] L. Yu, X. Luo, C. Qian, S. Wang, and H. N. Leung, "Enhancing the description-to-behavior fidelity in android apps with privacy policy," *IEEE Transactions on Software Engineering*, vol. 44, no. 09, pp. 834–854, sep 2018.

[28] L. Yu, T. Zhang, X. Luo, and L. Xue, "Autoppg: Towards automatic generation of privacy policy for android applications," *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2015.

[29] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 1036–1046.

[30] P. Wijesekera, A. Baokar, L. Tsai, J. Reardon, S. Egelman, D. Wagner, and K. Beznosov, "The feasibility of dynamically granted permissions: Aligning mobile privacy with user preferences," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 1077–1093.

[31] T. T. Nguyen, D. C. Nguyen, M. Schilling, G. Wang, and M. Backes, "Measuring user perception for detecting unexpected access to sensitive resource in mobile apps," *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021.

[32] Z. Zhang, Y. Feng, M. D. Ernst, S. Porst, and I. Dillig, "Checking conformance of applications against GUI policies," in *ESEC/FSE 2021: The ACM 29th joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Athens, Greece, Aug. 2021.

[33] W. Cao, C. Xia, S. T. Peddinti, D. Lie, N. Taft, and L. M. Austin, "A large scale study of user behavior, expectations and engagement with android permissions," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 803–820. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/cao-weicheng

[34] V. K. Malviya and A. Gupta, "Deep-learning-based malicious android application detection," in *Machine Vision and Augmented Intelligence—Theory and Applications*. Singapore: Springer Singapore, 2021, pp. 275–286.

[35] V. Sihag, M. Vardhan, P. Singh, G. Choudhary, and S. Son, "De-lady: Deep learning based android malware detection using dynamic features," *Journal of Internet Services and Information Security*, vol. 11, 05 2021.

[36] O. N. Elayan and A. M. Mustafa, "Android malware detection using deep learning," *Procedia Computer Science*, vol. 184, pp. 847–852, 2021, the 12th International Conference on Ambient Systems, Networks and Technologies (ANT) / The 4th International Conference on Emerging Data and Industry 4.0 (EDI40) / Affiliated Workshops. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050921007481

[37] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im, "A multimodal deep learning method for android malware detection using various features," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 773–788, 2018.

[38] M. Ashawa and S. Morris, "Android permission classifier: a deep learning algorithmic framework based on protection and threat levels," *Security and Privacy*, vol. 4, 2021.

[39] L. Bao, D. Lo, X. Xia, and S. Li, "Automated android application permission recommendation," *Science China Information Sciences*, vol. 60, no. 9, p. 092110, 2017. [Online]. Available: https://doi.org/10.1007/s11432-016-9072-3

[40] S. Niu, R. Huang, W. Chen, and Y. Xue, "An improved permission management scheme of android application based on machine learning," *Security and Communication Networks*, vol. 2018, p. 2329891, 2018. [Online]. Available: https://doi.org/10.1155/2018/2329891

[41] D. Kong and H. Jin, "Towards permission request prediction on mobile apps via structure feature learning," in *SDM*, 2015.

[42] M. Ricardo, C. Mariana, V. Jo˜ao, P., and B. Alastair, R., "Enhancing user privacy in mobile devices through prediction of privacy preferences," ser. ESORICS 2022, Copenhagen, Denmark, 2022.

[43] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, "Software vulnerability detection using deep neural networks: A survey," *Proceedings of the IEEE*, vol. 108, no. 10, pp. 1825–1848, 2020.

[44] D. Zhu, H. Jin, Y. Yang, D. Wu, and W. Chen, "Deepflow: Deep learning-based malware detection by mining android application for abnormal usage of sensitive data," in *2017 IEEE symposium on computers and communications (ISCC)*. IEEE, 2017, pp. 438–443.

[45] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," in *The Network and Distributed System Security Symposium*, 2018.

[46] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, "Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection," *Information and Software Technology*, vol. 136, p. 106576, 2021.

[47] S. S. E. G. at Paderborn University and F. IEM. (2022) Flowdroid data flow analysis tool. [Online]. Available: https://github.com/secure-software-engineering/FlowDroid

[48] "Soot program analysis framework," 2021, http://soot-oss.github.io/soot/.

[49] "Androwarn: Yet another static code analyzer for malicious android applications," 2019, https://github.com/maaaaz/androwarn.

[50] G. Kim. (2022) Gator: Program analysis toolkit for android. [Online]. Available: http://web.cse.ohio-state.edu/presto/software/gator

[51] "Axplorer mappings," https://github.com/reddr/axplorer,2022.

[52] Graphviz, "Dot language," https://graphviz.org/doc/info/lang.html.

[53] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad, "Collective classification in network data," *AI Magazine*, vol. 29, no. 3, p. 93, Sep. 2008. [Online]. Available: https://ojs.aaai.org/index.php/aimagazine/article/view/2157

[54] Y. Ma and J. Tang, *Deep Learning on Graphs*. Cambridge University Press, 2021.

[55] H. Cai, V. W. Zheng, and K. C.-C. Chang, "A comprehensive survey of graph embedding: Problems, techniques, and applications," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 9, pp. 1616–1637, 2018.

[56] "Networkx network analysis," 2021, https://networkx.org/documentation/stable/tutorial.html.

[57] "Gl2vec: Graph embedding enriched by line graphs with edge features," 2019, https://link.springer.com/chapter/10.1007/978-3-030-36718-3_1.

[58] "Karateclub whole graph embedding," 2021, https://karateclub.readthedocs.io/en/latest/modules/root.html.

[59] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, "graph2vec: Learning distributed representations of graphs," 2017. [Online]. Available: https://arxiv.org/abs/1707.05005

[60] "sklearn ensemble module," 2021, https://scikit-learn.org/stable/modules/ensemble.html.

[61] "Singapore smart nation apps," https://www.smartnation.gov.sg/community/apps-for-you, 2022.

[62] "Androzoo," https://androzoo.uni.lu, 2022.

[63] S. Xi, S. Yang, X. Xiao, Y. Yao, Y. Xiong, F. Xu, H. Wang, P. Gao, Z. Liu, F. Xu *et al.*, "Deepintent: Deep icon-behavior learning for detecting intention-behavior discrepancy in mobile apps," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2421–2436.

[64] "Android app permissions," https://developer.android.com/reference/android/Manifest.permission,2022.

[65] S. Y. K. Neo, "Graph-based malware detection on the android phones," 2014.

[66] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 590–604.

[67] Sophos, william. lee, and X.-W. Wu, "Cross-platform mobile malware : Write once , run everywhere," 2015.