ProMAL: Precise Window Transition Graphs for Android via Synergy of Program Analysis and Machine Learning

Changlin Liu1  Hanlin Wang1  Tianming Liu2  Diandian Gu3  Yun Ma3
Haoyu Wang4  Xusheng Xiao1
1Case Western Reserve University, 2Monash University
3Peking University, 4Beijing University of Posts and Telecommunications
1{cxl1029,hxw458,xusheng.xiao}@case.edu, 2Tianming.Liu@monash.edu, 3{gudiandian1998,mayun}@pku.edu.cn,
4haoyuwang@bupt.edu.cn

ABSTRACT
Mobile apps have been an integral part in our daily life. As these apps become more complex, it is critical to provide automated analysis techniques to ensure the correctness, security, and performance of these apps. A key component for these automated analysis techniques is to create a graphical user interface (GUI) model of an app, i.e., a window transition graph (WTG), that models windows and transitions among the windows. While existing work has provided both static and dynamic analysis to build the WTG for an app, the constructed WTG misses many transitions or contains many infeasible transitions due to the coverage issues of dynamic analysis and over-approximation of the static analysis. We propose ProMAL, a “tribrid” analysis that synergistically combines static analysis, dynamic analysis, and machine learning to construct a precise WTG. Specifically, ProMAL first applies static analysis to build a static WTG, and then applies dynamic analysis to verify the transitions in the static WTG. For the unverified transitions, ProMAL further provides machine learning techniques that leverage runtime information (i.e., screenshots, UI layouts, and text information) to predict whether they are feasible transitions. Our evaluations on 40 real-world apps demonstrate the superiority of ProMAL in building WTGs over static analysis, dynamic analysis, and machine learning techniques when they are applied separately.

CCS CONCEPTS
• Theory of computation → Program analysis; • Software and its engineering → Software testing and debugging; • Computing methodologies → Machine learning.

KEYWORDS
mobile apps; window transition graph; static analysis; deep learning

1 INTRODUCTION
Mobile applications (i.e., apps) have become an integral part of our daily life, from entertainment, travel, education, and even to business [12, 53]. Thus, it is critical to improve the quality and reliability of these apps by developing automated analysis techniques to ensure the correctness, security, and performance of these apps [19, 26, 41, 47, 47, 54, 69, 70]. As a key component for these automated analysis techniques, we focus on creating a graphical user interface (GUI) model of an app, i.e., a window transition graph (WTG). In WTG, nodes represent windows and edges represent transitions between windows, triggered by callbacks executed in the GUI thread. For example, clicking a button in the GUI will result in the execution of a callback that changes the screen to display another window. WTG can be directly used for understanding, testing, and exploring apps’ behaviors [7, 26, 54]. It can also assist static analyses, such as detecting security-sensitive behaviors and other non-functional properties like energy efficiency [9, 10, 63, 65].

While considerable research efforts have been spent to construct WTGs [26, 41, 47, 54, 70] via either static analysis techniques or dynamic analysis techniques, it is still challenging to obtain an accurate WTG. On one hand, dynamic analysis such as dynamic exploration [26, 41, 54] is precise in identifying transitions between windows, but this type of techniques suffer from the notorious coverage problem as other dynamic analysis techniques [14, 64, 66]. On the other hand, while static analysis that models the GUI objects, events, and callbacks [47, 69, 70] shows promising results in constructing a more comprehensive WTG, the imprecision in reference analysis and over-approximation in computing data flows often result in infeasible transitions. For example, if the imprecision of the reference analysis causes several buttons to be aliases with each other, then a transition triggered by one button will also result in the model to include incorrect transitions from the other buttons. Such incorrect transitions will further cause the imprecision for the downstream analysis such as control and data flow analysis [47, 69].

To address these challenges, we propose a novel “tribrid analysis” approach, ProMAL, that synergistically combines static/dynamic program analysis and machine learning techniques to construct a WTG for an app. In particular, ProMAL aims to benefit from the precise analysis from dynamic analysis, and at the
same time mitigate the low coverage of dynamic analysis and the imprecision caused by the static analysis. ① Specifically, ProMal first combines static analysis and dynamic analysis: ProMal applies static analysis to construct a WTG, and then runs dynamic analysis to verify the detected transitions. Due to the coverage issues of dynamic analysis and over-approximation of the static analysis, it is expected that a substantial amount of transitions cannot be verified by the dynamic analysis. ② Instead of including all these unverified transitions in the WTG, which potentially will generate many infeasible transitions, ProMal further uses a machine learning technique, window transition prediction, that leverages the features for the unverified transitions (e.g., screenshots and text) to predict which transition to include. Our novel techniques (① for verifying static WTGs using dynamic WTGs and ② for predicting unverified transitions) allowing ProMal to construct a more precise WTG that can benefit downstream analysis techniques.

Existing static analysis techniques [47, 69, 70] are effective in detecting transitions among windows, but are limited in detecting transitions among dialogs. ProMal improves the existing static analysis by modeling the dialog builder APIs and detects transitions buried in the callbacks of dialogs to identify the transitions from dialogs to other windows and dialogs. As dynamic analysis [26, 41, 54] typically models the window transitions differently from the static analysis, ProMal instruments the app under analysis, which will record the information that can be used to align the dynamic WTG with the static WTG. This includes the identifiers of GUI widgets (i.e., XPath) and interaction events. Moreover, the instrumentation will collect the runtime information, including screenshots of the windows, the view tree① [23], and the text shown in the windows. The runtime information is later used by the machine learning techniques to predict unverified transitions. For the windows found by static analysis but dynamic analysis cannot reach due to coverage issues, ProMal uses static UI rendering based on ADT (Android Developer Tools) [3, 32] to render the UI layouts for obtaining the screenshots and the view tree, and applies static analysis on the UI layout file to obtain the text.

To predict whether unverified transitions should be included in the WTG, we construct a model of window transition prediction based on machine learning techniques. Given a GUI widget in a window and another window, the model first extracts features of them to learn their low-dimensional representations, which are then fed into a link predictor to estimate the likelihood of whether these exists a transition between the widget and window. However, due to the large amount of parameters in the prediction model, it is difficult to directly train the model since obtaining a large amount of manually labelled WTGs is infeasible. To address this issue, we adopt a two-phase training models: (1) pre-training on the dynamic WTGs collected by dynamic program analysis from a large amount of apps, and (2) fine-tuning on the manually labelled WTGs from a small set of apps. The essence of the design lies in getting a model general enough to accurately predict window transitions.

We implement ProMal upon Gator [47, 69, 70] and Paladin [41], and evaluate ProMal on a diversified set of real apps (40 apps with ~ 2.5 million LOC). These apps have non-trivial WTGs of different sizes (# of edges ranging from 5 to 538). We apply Paladin to construct WTGs for these apps, and compare the WTGs with the groundtruth WTGs. We pre-train our prediction model on 1,625 apps explored by Paladin, and fine-tune the model using 90% of the apps in our evaluation dataset. We perform 10-fold validation to predict the transitions for each app. The results show that ProMal effectively identifies the feasible transitions among the windows in constructing WTGs, achieving a precision of 90.18%, a recall of 79.69%, and a F1-score of 82.82% on average. Moreover, the WTGs built by ProMal achieves a significantly better F1-score than the WTGs built by Gator (46.24%) and the combined WTGs built by Gator and Paladin (61.93%). These results reveal the limitations of static analysis in modeling windows/callbacks for diversified real apps, and demonstrate the effectiveness of ProMal in using a synergy approach of program analysis and machine learning.

This paper makes the following main contributions:

- A novel approach, ProMal, that synergistically combines static/dynamic analysis and machine learning to construct WTGs for Android apps.
- A novel static analysis technique that builds static WTGs with transitions among windows and dialogs.
- A novel dynamic analysis technique that instruments apps under analysis and leverages app exploration techniques to build dynamic WTGs and collect runtime information for window transition prediction.
- A machine-learning model to predict window transitions, which can be pre-trained on apps explored by dynamic analysis and fine-tuned using labelled WTGs to improve the performance.
- An evaluation on a diversified set of Android apps to demonstrate the effectiveness of ProMal. The tool and the results are available at the project website [4].

2 BACKGROUND

In an Android app, an activity provides a window to draw the GUI [23]. A GUI consists of GUI widgets (e.g., buttons and text boxes) and layout models (e.g., linear layout) that describe how to arrange UI widgets. Each GUI widget can respond to several events, where each event triggers a sequence of callbacks. For example, the click event in a button corresponds to clicking the button, and it triggers the event handler callback that is registered to the click event, such as onClick. A callback can open a new window. For example, this can be done via calling the API startActivity. When a new window is opened, it causes a window transition. Besides widget event callbacks, hardware events (e.g., pressing BACK or HOME button) can also trigger callbacks to cause window transitions. In particular, Android maintains a back stack that stores the order of the opened window, so that it can be used to decide which window to return to when the BACK button is pressed.

Window transition graph (WTG) is a type of GUI model that represents window transitions. Existing work [47, 69, 70] defines a WTG as a directed graph, where nodes represent windows and edges represent transitions. The edges in WTG are annotated with three types of labels (i.e., \( e, \delta, \sigma \)), where \( e \) represents the event to trigger the transition, \( \delta \) represents a sequence of window stack

①A tree structure shows that hierarchy of the GUI widgets and the layout containers (e.g., LinearLayout and RelativeLayout).
operations that push or pop the window to the window stack$^2$, and $\sigma$ represents the sequence of callbacks for the transition. Besides widget events such as click events, the model further supports five default events that corresponds to pressing the BACK, HOME, POWER, MENU buttons and rotating the phone. A series of static analysis techniques are then developed to identify GUI widgets, associate their callbacks, and build a static WTG. While dynamic analysis such as app exploration [26, 40, 41, 54] usually does not explicitly build a WTG, they provide their own models to represent UI states, which can be considered as another type of WTGs with different representations for windows and their transitions.

Applications of WTGs. WTGs can be used to improve various types of software analysis, such as testing, security vetting, and performance profiling [11, 56, 60, 62, 63, 68]. Wu et al. [62] generates GUI tests based on the paths in the WTGs and analyzes the UI callbacks and activity life cycle methods to examine sensors that are not properly released. Yang et al. [68] uses a WTG to determine which activities can reach more other activities to prioritize test exploration. Another GUI testing tool [11] uses a statically computed WTG to guide which widget to trigger when the other exploration rules fail. Besides testing, WTGs are also used for performance profiling and security vetting. Wang et al. [60] profile potential resource demanding tasks in the UI thread by using WTGs to identify callbacks that trigger “janky” operations and the window transition sequences that trigger such callbacks. Tang et al. [56] build a UI-oriented program dependence graph, which is essentially a WTG, to discover link hijacking vulnerabilities. All these applications rely on WTGs built explicitly or implicitly, and thus it is crucial to improve the precision of the built WTGs for improving the effectiveness of these applications.

3 MOTIVATION EXAMPLE

Figure 1 shows the UI (after translation) of an example window transition of the app “Subscriber Assistant Application”$^3$, which allows users to subscribe various services in Russia. When the highlighted button is clicked, the UI transits from the activity Root to the activity Menu, resulting in a window transition. We can construct a WTG for this app to model these transitions, where 8 nodes are used to represent the activities and 125 edges are used to represent the transitions among them. Due to space limit, Figure 2 shows the partial WTG of the app. There are 5 nodes and 12 edges in the WTG. The nodes show the name of the activity (e.g., a0: Root and a1: Favorites). The labels on the edges represent the events to trigger the transition (e.g., a3: btn, click means that after clicking the button, the active window of app will transit from the activity Root to the activity Favorites).

To obtain this WTG, ProMal first applies both static and dynamic WTG analysis to obtain the static WTG and the dynamic WTG, respectively. Then, ProMal aligns the static WTG and the dynamic WTG to form a single WTG. As shown in Figure 2, the solid edges represent matching edges between the static WTG and the dynamic WTG, i.e.,, verified edges. Due to the over-approximation of static WTG analysis and the coverage issue of the dynamic WTG analysis, there always will be some edges in the static WTG that cannot be verified by the dynamic WTG. The one dashed edge in Figure 2 represents such kind of unverified edge. To address this problem, ProMal further uses window transition prediction to predict whether the unverified edges are likely to be transitions or not. To do so, ProMal collects the screenshots and the text information

---

$^2$Window stack is a generalization of Android back stack that includes more types of windows (dialogs and menus) and models the changes of the stack.

$^3$Package name of the app is “com.olsoft.sa.ntvplus”.
of the activities (as shown in Figure 1) and also the structural information of the UI layout, i.e., the view tree. Figure 3 shows the partial view tree of the activity root. As we can see, the view tree shows the hierarchy of the UI widgets (e.g., ListView) and the layout container (e.g., LinearLayout), and also the boundary of each UI widget (i.e., represented using screen coordinates). Since Android uses XML layout files to build view trees at runtime, XPath [1] can be used to locate the UI widget and the layout container. For example, the XPath shown at the bottom of Figure 3 points to a LinearLayout that can be clicked, and the "Bounds" shows its boundary in the UI using screen coordinates. With the screenshots and the text information of the activities and the view tree as input, the window transition prediction correctly predicts the transition from the activity Menu to the activity OptionMenu in Figure 1, enhancing the WTG constructed by the static and dynamic WTG analysis.

4 OVERVIEW

Figure 4 shows the workflow of ProMAL. ProMAL consists of three major components: static WTG analysis, dynamic WTG analysis, and window transition prediction. ProMAL accepts an Android app APK file as input and applies both static and dynamic WTG analysis to obtain the static and dynamic WTGs, respectively. The static WTG analysis performs static analysis on the GUI layout files and the code of the APK file and outputs a static WTG. The dynamic WTG analysis consists of two steps: it first performs app instrumentation, which adds the code component into the APK file to collect runtime information, and then applies dynamic app exploration to explore apps' behaviors for building the dynamic WTG. ProMAL then aligns the static and dynamic WTGs to obtain the edges (i.e., window transitions) that are verified by the dynamic WTG analysis (i.e., verified edges). For the unverified edges, ProMAL uses the window transition prediction to predict whether the unverified edges are likely to be feasible transitions. The window transition prediction trains its model on a set of training APK files, and uses the runtime information (i.e., screenshots, text information, and UI layouts) to predict window transitions. As unverified edges are not explored by the dynamic analysis, ProMAL statically renders the GUIs involved in the unverified edges for obtaining the information required for the prediction. The output WTG contains only the verified edges and the edges predicted to be feasible edges.

5 DESIGN OF PROMAL

In this section, we first provide the formal definition of window transition graph (WTG), and then describe the three major components of ProMAL: static WTG analysis, dynamic WTG analysis, and window transition prediction.

5.1 Definition of WTG

We represent a WTG as a directed graph $G = (Win, E, \epsilon)$, where $Win$ represents a set of nodes where each node represents a window of an app, $E \subseteq (Win \times Win)$ is the set of edges that represent transitions among windows, and $\epsilon : E \rightarrow Eset$ represents the edge labels that describe the events $Eset$ that cause the transitions.

Window. We consider 3 categories of windows that users can interact with as a node in a WTG: activities, menus, and dialogs. An activity is often presented to the users as a full-screen window, serving as the build block of an app’s GUI. Menus include OptionMenu which is associated with activities and ContextMenu which is associated with GUI widgets. Dialogs are short-lived windows that often need user actions to proceed to the next window.

Edge (Transition). An edge $e = (win_i, win_j)$ represents a transition from a source window $win_i$ to a target window $win_j$, and the labels on the edges describe the events that cause the transition, such as a button click. Without loss of generality, we treat back edges as the other edges but with a special “back” label, which makes it easier to match the back edges identified in the static WTG with the dynamically observed ones. Moreover, we exclude loops in $E$. A loop is an edge $e_l = (win_i, win_i)$ that points to itself, i.e.,, $win_i = win_i$. For example, when the user clicks a button representing numbers in a calculator app, which does not result in a window transition but stays at the same window, resulting in a loop. As GUI widgets that do not cause inter-window transitions will result in loops, these edges are of a large amount. Thus, adding loops to $E$ only complicates a WTG without providing more useful information.

Event. An event $evt = (w, t)$ is a label associated with an edge $e = (win_i, win_j)$, where $w$ represents a GUI widget in $w$, and $t$ is the type of this event (e.g., $w$ is a button and $t$ is “long_click”). We model two types of events: widget events and default events. Widget events correspond to the interactions with a GUI widget (e.g., clicking a button), which are categorized into two groups: (1) click events, including touch, select, click, item_clicked, and item_selected, and (2) long click events, including long_click and item_long_click. We exclude several widget events because they mainly cause loops, such as scroll, drag, focus_changed, and enter_text. In fact, in our evaluation dataset, all 181 edges corresponding to these events are loops. Default events correspond to the interactions with the physical buttons or rotating the device. We focus on the events caused by pressing the back button and the menu button, and exclude events caused by rotating the phone and pressing the home button and the power buttons since these excluded default events only cause loops and app switches. In our dataset, 1,116 out of 1,430 edges related to these events are loops, and the remaining 314 edges represent app switches. For example, when an app shows a menu or a dialog and the user presses the home button, the phone exits the app and shows the home screen; when the user goes back to the app, the app shows the parent activity of the menu or the dialog.
While it seems like a "transition" from the menu or the dialog back to the activity, it is in fact an app switch, which is generally not interesting in app explorations or testing [26, 41, 54, 72]. Our model currently does not include System events since they usually do not trigger window transitions but rather changing the states of a window. System events correspond to changes of system states, such as receiving new messages and volume adjustment [5, 17, 42, 45, 54, 59]. These events are of a huge amount and can significantly damper the performance of the testing tool, and thus they are randomly injected during testing [45, 54].

5.2 Dynamic WTG Analysis
Dynamic WTG analysis instruments apps and leverages app exploration techniques to automatically explore apps’ behaviors and collect the runtime information for building dynamic WTGs and predicting window transitions. We next describe the app instrumentation and runtime information collection.

App Instrumentation. ProMal instruments an app to record widget interactions and window transitions. Specifically, based on our WTG definition, ProMal monitors two major types of interaction events:
- **Widget Events**: ProMal records the click events, the click coordinates, and the GUI widgets.
- **Default Events**: ProMal records the default events for pressing the BACK and the MENU buttons.

To correctly identify window transitions, the instrumentation records the foreground activity before each interaction, and leverages the changes of the foreground activity to identify the source window and the target window when a transition happens. During the exploration, we collect the attributes of the visited windows (e.g., titles and texts), the call stacks of their parent methods, and the screenshots of dialogs and menus.

Since some GUI widgets may not possess widget IDs, ProMal further uses coordinates and XPath to identify GUI widgets in a window, as illustrated in Figure 3. ProMal hooks the `dispatchTouchEvent` API to obtain the screen coordinates for each interaction (e.g., clicking a button), and leverages UIAutomator [25] to obtain the information of the GUI state after the interaction, which includes a view tree and a screenshot. As Android’s GUI is rendered based on an XML layout file, the rendered GUI state can be represented as a view tree, where the root element is a layout container such as `LinearLayout`. Based on this view tree, we can use XPath to describe the path from the root element to the XML element of the clicked GUI widget. Besides, a view tree provides the boundaries of each GUI widget, which enables ProMal to locate the GUI widgets in the view tree using the screen coordinates of an interaction, and then generate the XPath of the clicked widget accordingly. To ensure the interaction event is fully executed when we capture the GUI state of the activity, it is in fact an app switch, which is generally not interesting in app explorations or testing [26, 41, 54, 72]. Our model currently does not include System events since they usually do not trigger window transitions but rather changing the states of a window. System events correspond to changes of system states, such as receiving new messages and volume adjustment [5, 17, 42, 45, 54, 59]. These events are of a huge amount and can significantly damper the performance of the testing tool, and thus they are randomly injected during testing [45, 54].

5.3 Static WTG Analysis
Static WTG analysis applies static analysis to identify windows, GUI widgets, and the transitions among the windows. Based on our definition, three types of windows are considered: activities, dialogs, and menus. Note that we consider both the classes “ContextMenu” and “OptionsMenu” as menus. ProMal first applies the existing static analysis technique (i.e., Gator) [47, 70] to identify edges among windows as we defined in 5.1. It performs a constraint graph based reference analysis to model Android GUI related objects (e.g., Activities, Views, and callback listeners) and their association relationship. It then builds a WTG based on the analysis of the GUI event callbacks and the window lifecycle callbacks. Besides modeling GUI related objects, ProMal extracts GUI widget information (i.e., title and text) from the view tree, which is later used for matching the dynamic WTGs. For events that trigger transitions, our static analysis focuses on widget events (i.e., “Click” and “Long Click” events) and default events (i.e., pressing the back and the menu buttons).

**Dialog Transitions.** Existing static analysis [47, 69] mainly identifies transitions from activity window to dialogs, but fails to identify transitions from the dialogs to other windows. The major reason is that these transitions are often buried in the callbacks of dialogs, which are defined using specific APIs in dialog builders (e.g., the `setItems` API of `AlertDialog.Builder`). To identify these transitions, we extend the static analysis techniques to identify these APIs, and associate the callbacks built through these APIs to the dialog constructed by the dialog builders. Specifically, our extended static analysis techniques identify the following dialog transitions:
- **To Other Dialogs**: the analysis examines whether a known dialog allocation API call is found in the associated callbacks.
- **To Host Activities**: executing APIs such as `dismiss` or `cancel` and registering a `null` or `nop` handler will trigger a transition to the host activity of the dialog. To detect such transitions, our analysis first leverages the dynamic runtime information collected in Section 5.2 to identify the host activities for all the dialogs; if a dialog is not covered during dynamic exploration, our analysis uses the callback registration sites identified by the static analysis techniques to infer the activity.
- **To Other Activities**: the analysis examines the activity transition calls (e.g., `startActivity()` and `startActivityForResult()`) and the values of their `Intent` arguments to identify the other activities.
- **To Previous Activities**: calling APIs like `Activity.finish` will cause transitions to the previous activity. These transitions are inferred based on the collected dynamic exploration traces.
5.4 Alignment of Static and Dynamic WTG

After obtaining the static WTG and the Dynamic WTG of an app, ProMAL aligns them by matching each edge in the static WTG to an edge in the dynamic WTG. We next describe the detailed steps.

Matching Windows. ProMAL matches windows using different attributes based on their types: (1) for activities, ProMAL checks their activity names; (2) for menus, ProMAL checks the names of the activities that create the menus; (3) for dialogues, ProMAL checks the call stack of the functions that creates the dialogs.

Matching Events and GUI Widgets. To match the event on an edge, we first check if the event types are identical. For widget events that associate with a certain GUI widget, ProMAL tries to obtain the widget ID in both the static WTG and the dynamic WTG to match the widgets. However, the edges in the static WTGs do not always possess a widget ID. Sometimes they can only provide a class name of the GUI widgets or even provide nothing to infer the associated widgets. This imprecision of static analysis makes it impossible for users to pinpoint the widgets, and hence they are regarded as unmatched.

The edges in the static WTGs that cannot be matched will be subject to further machine learning prediction.

5.5 Window Transition Prediction

The window transition model is used to predict the unverified edges in the static WTGs. As shown in the example of Figure 5, the window transition model consists of the embedding models (the widget embedding model and the window embedding model) and the link predictor. Given a pair of a GUI widget and a window, ProMAL first extracts the features from them, such as screen snapshot and text, and then feeds these features into the embedding models to obtain the embedding vectors of the GUI widget and the window. The link predictor then uses the two embedding vectors to tell how likely there is a link between them, which is represented by \( \Psi(\phi_w(w), \phi_a(a)) \). The score is computed by using the following one-hot embedding is used to represent the tag. Besides the tags, the positions and sizes of the GUI widgets may also be used to infer the functionality of widgets. To encode all the GUI widgets in a window, ProMAL traverses the GUI tree via in-order tree traversal to generate a widget sequence, and adapts LSTM [22, 29, 30] to learn the representation of the widget sequence. Figure 5 shows how the tags and other information in a GUI tree are unrolled to a sequence.

The representations of these three features are then concatenated and fed to a fully connected layer (FC) to generate the window embedding vector.

Widget Embedding Model. To generate an embedding vector for a GUI widget, besides the features of the source window, the widget embedding model also extracts the features of a GUI widget, including the widget screenshot, the text, and the GUI properties. The feature extraction for the source window adopts the same approach as the window embedding model. We next describe the other three features:

- **Widget Screenshot**: The widget embedding model uses a block of DenseNet for modeling the screenshot of the source window, and another block of DenseNet for modeling the screenshot of the GUI widget, as shown in the red frame in Figure 5.
- **Text**: The widget embedding model segments both the texts in the source window and the GUI widget and feed both sequences of words to the Word2Vec model to generate two vectors. As shown in Figure 5, the widget embedding model computes the word representation of the text "Does not repeat" in the GUI widget as well as the average embedding of every word in the source window.
- **GUI Property**: The widget embedding model encodes the GUI tree using a similar approach as the window embedding model. Apart from the widget sequence of the source window, it also extracts the tag, the size, and the position of the GUI widget, and feeds these features to a fully connected layer to generate a vector, as shown in Figure 5.

The feature vectors of the source window and the GUI widget are concatenated and fed into a fully connected layer to generate the widget embedding vector.

Link Predictor. The widget embedding and window embedding generated by the embedding models will be fed to the link predictor to infer whether there is a link between the widget-window pair. The link predictor is designed by leveraging the neural tensor network (NTN) [52], which relates the two inputs (i.e., the widget embedding and window embedding vectors) multiplicatively instead of only implicitly through the non-linearity as with the standard neural networks where the entity vectors are simply concatenated. Thus, it provides a more powerful way to infer the relationship between entities than a standard neural network layer.

Let \( w \) represent a GUI widget and \( a \) represent a window of an app activity. \( \phi_w \) is the widget embedding model and \( \phi_a \) is the window embedding model. The link predictor computes a score of how likely there is a link between them, which is represented by \( \Psi(\phi_w(w), \phi_a(a)) \). The score is computed by using the following
Figure 5: Illustration of how the window transition model predicts whether interacting with a GUI widget in a source window causes a transition to a target window. The figure shows part of the extracted features for the GUI widget, the source window, and the target window, and illustrates how they are processed in the embedding models to produce the embedding vectors, which are used by the link predictor to make the prediction.

function:

\[
\Psi(\phi(w), \phi_d(a)) = u^T f(\phi(w)^T W_1^{[1]} \phi_d(a) + V_R \phi(w) + b_R) + b_R
\]

(1)

where \( f = tanh \) is a standard non-linearity applied element-wise and \( W_R^{[1]} \in \mathbb{R}^{d \times nd} \) is a tensor. \( \phi(w)^T W_1^{[1]} \phi_d(a) \) is a bilinear tensor product and results in a vector \( d \in \mathbb{R}^d \), where each entry is computed by one slice \( m = 1, \ldots, k \) of the tensor: \( \phi(w)^T W_1^{[1]} \phi_d(a) \). The other parameters for relation \( R \) are the standard form of a neural network: \( V_R \in \mathbb{R}^{k \times 2d} \) and \( U \in \mathbb{R}^k, b_R \in \mathbb{R}^k \) [52].

Two-Phase Training. Due to the large amount of parameters in the model, we have to use a correspondingly large amount of widget-window pairs as the training data, which are infeasible to be manually labeled. To address this challenge, we adopt a two-phase training process. We first apply dynamic app exploration techniques [26, 41, 54] to automatically explore a large number of apps dynamically and pre-train the model based on the dynamically observed transitions. The variety of apps makes the embedding models general enough. As these transitions bias towards the transitions that can be easily found by dynamic analysis, we further fine tune the pre-trained model based on the manually identified window transitions from a smaller set of apps. In the fine tuning process, the parameters of the embedding models are frozen (i.e., weights and biases), meaning that these parameters do not change, while the parameters of the link predictor are set trainable to adapt the patterns related to window transitions.

6 EVALUATION

In this section, we seek to evaluate the effectiveness of ProMal to construct window transition graphs for real-world Android apps. We implement ProMal in Java. ProMal uses GATOR [47, 69, 70], a state-of-the-art static Android GUI analysis tool, to build static WTGs, and uses PALADIN [41], a state-of-the-art app exploration tool, to build dynamic WTGs. ProMal also uses Xposed [2] to instrument apps for collecting window transitions and other runtime information. The window transition prediction is implemented using Keras [13]. Specifically, we aim to answer the following research questions:

- **RQ1**: How effectively can ProMal build the WTGs?
- **RQ2**: How effectively can ProMal improve over GATOR and PALADIN?
- **RQ3**: How effectively can the NTN model and the two-phase training improve ProMal’s window transition prediction?
6.1 Subjects and Evaluation Setup

We use a diversified set of apps from Google Play [24] as our evaluation subjects. These apps are from 14 categories, such as Game, Entertainment, and Education. We also include popular open-sourced apps from F-Droid [20] in our evaluation subjects. Once we downloaded an app, we applied GATOR to build static WTGs and excluded the apps with less than 3 windows, since their WTGs are fairly simple and will not be proper subjects for assessing the effectiveness of ProMAL. In total, we examined 4,326 apps and got 2,216 apps with at least 3 windows. As shown in Figure 2, a node (window) may have multiple edges (transitions) to another node in WTGs, making the WTGs more complex than the WTGs that have one or two transitions among windows. Thus, we use density to represent the complexity of WTGs, where the density is computed using the number of edges divided by the number of nodes in static WTGs. Note that the edges in static WTGs may be infeasible, but they can still form a reasonable estimation of the actual WTG. Based on the density, we divided the apps into 7 groups equally, and randomly sample a subset from each group to evaluate ProMAL on the WTGs of different complexities. As there is no publicly available groundtruth WTG for these apps, we need to construct the groundtruth WTGs by manually exploring the apps and inspecting their source code. Because the number of edges grows drastically as the complexity of the App grows, collecting the groundtruth requires a non-trivial effort. Within our affordable efforts, we randomly chose 5 apps from each group as our evaluation subjects. Together with the 5 apps chosen from F-Droid, we have 40 apps in 8 density groups. These apps also belong to different popular categories (e.g., sports, game, tools). The total LOC of these apps is \( \sim 2.5 \) million. The summary of the apps is shown in Table 1, and more details can be found on our website [4].

We then applied PALADIN and ProMAL on these 40 apps to build dynamic WTGs and performed window transition predictions to build the optimized WTGs. PALADIN is a dynamic app exploration tool that models view trees as UI states to avoid visiting the same UI states during the exploration. It triggers actionable widgets in a Depth-First-Search manner to exercise as many behaviors as possible. Among all the apps analyzed by PALADIN, the minimum running time is 1.35 minutes, and the maximum time is around 97 minutes. The average running time for all the apps is about 15.8 minutes per app.

### Table 1: 40 apps used in our evaluations. The 35 real apps are sorted by their densities and divided evenly in 7 groups. F-Droid is the last group.

<table>
<thead>
<tr>
<th>Group</th>
<th>Density</th>
<th>LOC</th>
<th># Nodes</th>
<th># Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 1</td>
<td>1.5–3.25</td>
<td>33,048</td>
<td>28</td>
<td>61</td>
</tr>
<tr>
<td>Group 2</td>
<td>3.3–4.6</td>
<td>46,508</td>
<td>28</td>
<td>59</td>
</tr>
<tr>
<td>Group 3</td>
<td>5.2–7.8</td>
<td>78,217</td>
<td>32</td>
<td>83</td>
</tr>
<tr>
<td>Group 4</td>
<td>8.8–12.7</td>
<td>344,144</td>
<td>40</td>
<td>136</td>
</tr>
<tr>
<td>Group 5</td>
<td>14.0–23.6</td>
<td>389,586</td>
<td>72</td>
<td>186</td>
</tr>
<tr>
<td>Group 6</td>
<td>25.5–37.4</td>
<td>823,086</td>
<td>76</td>
<td>246</td>
</tr>
<tr>
<td>Group 7</td>
<td>37.6–155.6</td>
<td>55,259</td>
<td>43</td>
<td>122</td>
</tr>
<tr>
<td>F-droid</td>
<td>1.5–35.8</td>
<td>714,044</td>
<td>50</td>
<td>178</td>
</tr>
<tr>
<td>Total</td>
<td>–</td>
<td>2,483,892</td>
<td>369</td>
<td>1,071</td>
</tr>
</tbody>
</table>

6.2 RQ1: Overall Effectiveness

Table 2 shows the details of the WTGs built by ProMAL for each app. Column “# Nodes” shows the number of nodes in the WTG. Column “# Edges” shows the number of edges in the WTG. Column “Prec. (Edges)” and Column “Recall (Edges)” shows the precision and the recall of the detected edges. As we can see, on average,
Table 2: Details of the WTGs built by ProMal

<table>
<thead>
<tr>
<th>App Group</th>
<th># Nodes</th>
<th># Edges</th>
<th>Prec. (Edges)</th>
<th>Recall (Edges)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 1</td>
<td>25</td>
<td>56</td>
<td>89.00%</td>
<td>92.86%</td>
</tr>
<tr>
<td>Group 2</td>
<td>24</td>
<td>54</td>
<td>100.00%</td>
<td>92.95%</td>
</tr>
<tr>
<td>Group 3</td>
<td>28</td>
<td>63</td>
<td>97.33%</td>
<td>75.40%</td>
</tr>
<tr>
<td>Group 4</td>
<td>34</td>
<td>122</td>
<td>81.29%</td>
<td>71.08%</td>
</tr>
<tr>
<td>Group 5</td>
<td>52</td>
<td>148</td>
<td>85.98%</td>
<td>73.59%</td>
</tr>
<tr>
<td>Group 6</td>
<td>51</td>
<td>155</td>
<td>82.36%</td>
<td>59.34%</td>
</tr>
<tr>
<td>Group 7</td>
<td>36</td>
<td>107</td>
<td>90.37%</td>
<td>80.47%</td>
</tr>
<tr>
<td>F-Droid</td>
<td>27</td>
<td>154</td>
<td>97.09%</td>
<td>91.84%</td>
</tr>
<tr>
<td>Total</td>
<td>275</td>
<td>859</td>
<td>90.18%</td>
<td>79.69%</td>
</tr>
</tbody>
</table>

*Prec. and Recall are average values.

ProMal achieves a precision of 90.18%, a recall of 79.69%, and a $F_1$-score of 82.82%. In particular, ProMal achieves 97% precision in Groups 2 and 3, and 92% recall for Groups 1 and 2. It also achieves 91% precision and recall for the F-Droid group. These results show that ProMal is highly effective in identifying feasible edges in building WTGs.

Combination of Gator and Paladin. We compare the WTGs built by ProMal and the combination of Gator and Paladin. The results show that the WTGs built by combining the WTGs of Gator and Paladin achieve a precision of 93.59%, a recall of 88.47%, and a $F_1$-score of 61.93% on average. Since all the edges produced by Paladin are feasible edges, the low precision is caused by a huge number of infeasible edges introduced by Gator. For the 40 apps in our dataset, Gator yields 3,263 edges in total, while only 12% can be found in the WTGs built by Paladin. Thus, the majority of edges in Gator are further analyzed by the window transition prediction. For the 2,841 unverified edges, 47.1% (1,337) contains incomplete information (e.g., missing UI layout information or unable to locate the widget based on Gator’s UI widget information). Our model hence regards these edges as infeasible edges and predicts the remaining 1,504 edges. It is also noteworthy that 245 of the remaining edges from Gator are not specified with a widget ID. Instead, the widgets of these edges are often assigned to a whole window or an anonymous MenuItem, or a certain type of class name. To address such imprecision of Gator, ProMal considers all the clickable widgets that fit the widget information provided by Gator in the source window of these edges as the edges’ source widgets, and predicts whether the edges are feasible. For edges with back events, we consider them as feasible if a non-back edge from the target window to the source window can be found in the predicted WTG. Only the edges that are predicted to be feasible edges are kept in the final WTGs. With the window transition prediction, the precision of the built WTGs is significantly improved (from 53.95% to 90.18%), which results in a significant improvement of $F_1$-score (from 61.93% to 82.82%). These results clearly demonstrate the effectiveness of ProMal in using a synergy approach of program analysis and machine learning.

FPs and FNs. Due to the FPs and the FNs, ProMal achieves a relatively low precision in Groups 4, 5, 6 and a relatively low recall in Group 6 (i.e., more than 5% lower). Figure 6 illustrates how ProMal produces an FP and an FN. The window on the left is from the activity MainPage. ProMal correctly identifies a transition from MainPage to the activity Preference, resulted from clicking the “info” button (highlighted in green). However, ProMal also incorrectly identifies another transition to the activity Preference, resulted from clicking the “High” button (highlighted in red). This FP is caused by the patterns learned by the machine learning model. Figure 6 also includes a FN produced by ProMal. In the Preference activity, if the “More App” button is clicked, the app will transit to the More activity, which serves as a built-in browser and displays a website to users. Due to the drastic changes of the UI states, the window transition prediction model cannot infer any close relationship between the source window and the target window from their UI layouts and screen snapshots, and thus produces a FN.

Furthermore, the limitations of Gator’s model hinders the performance of ProMal. First, gator have trouble distinguishing widgets without widget IDs and will associate them with all possible handlers declared in the same method. This type of over-approximation makes ProMal generate extra edges. Second, a large number of false negatives are caused by Gator failing to identify dialog instances. Thus ProMal cannot detect transitions from these dialogs.

6.3 RQ2: Comparison with Paladin and Gator

Comparison with Gator Table 3 shows the details of the WTGs built by Gator for each app. As indicated by the results, many of the edges found by Gator are infeasible edges, i.e., FPs. These results show that the WTG built by Gator for each app is rather inaccurate, achieving a precision of 46.07%, a recall of 59.89%, and a $F_1$-score of 46.24%.

Upon further investigations, we find that Gator performs poorly on both precision and recall for two main reasons. The first reason is due to the imprecision of Gator. We empirically find out that Gator often yields multiple transitions with different target windows from the same UI widget, while most of them turn out to be infeasible. Figure 7 shows an example transition from the game app “com.twobitinc.cornholescore”. The left window is from the activity OptionsActivity, which shows 5 buttons for users to customize certain game parameters such as the color of the team and the scoring.
mode. When the "Set Color" button is clicked, the UI transits to the activity RgbPickActivity. However, due to the inprecision in associating widget IDs and event handler methods, Gator infers that clicking any of the button will transit to 3 different activities. In this way, Gator infers 2 infeasible edges for each of the 5 buttons, and results in 10 FP edges for a single activity, which significantly reduces Gator’s precision.

Second, Gator often fails to identify transitions between activities due to the limitation in modeling certain transitions. Figure 8 shows an example transition for the app "com.jacobsmedia.sparts". Upon clicking the “hockey” button, the app transits from the activity MainPage to another activity MainPage. But Gator fails to infer this edge. Actually, the built WTG for this app indicates that the activity MainPage, along with two other activities, have only transitions to themselves. However, there are in fact 9 edges among the three activities in the groundtruth WTGs, which cause 9 FNIs for Gator.

Compared to Gator, ProMal significantly improves both the precision and the recall in building WTGs, with the help of dynamic analysis and window transition prediction. The predictions on the unverified edges in total rule out 2,351 out of 2,399 FP edges, hence significantly improving the precision of ProMal by 95.02%, from 46.24% to 90.18%. By adding the 708 edges detected by Paladin and the 63 edges from Gator’s WTGs, the recall of the WTG built by ProMal is improved from 59.89% to 79.69%.

**Comparison with Paladin.** Table 4 shows the details of the WTGs built by Paladin. As Paladin adopts dynamic analysis, the precision in finding feasible edges is always 100%. On average, the WTGs built by Paladin achieve an average recall of 71.25% in finding feasible edges. We can observe that ProMal improves the built WTGs with the help of static analysis and window transition prediction. On average, ProMal improves the recall of Paladin by 8.44% and in turn improves the $F_1$-score by 2.35%. In particular, ProMal improves the recall significantly for Group 2 (from 76.36% to 92.95%) and Group 7 (from 68.23% to 80.47%). The increased recall will enable testing to cover more behaviors with tolerable extra efforts on the reported infeasible transitions, with the precision still being 90%.

### 6.4 RQ3: Window Transition Prediction

We first compare the performance of the alternative models for the link predictor, and then measure the improvement brought by the two-phase training.

**Comparison of Link Predictor Models.** We compare our NTN model with the bilinear model [33, 55], which is also an effective way to model the relationships between entities. We train both models using the window transitions collected from the dataset of 1,625 apps, which are automatically explored by Paladin. The results show that NTN achieves a precision of 96.97%, a recall of 95.26%, and a $F_1$-score of 95.73%, while the bilinear model only achieves a precision of 83.64%, a recall of 75.40% and a $F_1$-score of 73.98%. Thus, NTN is a better choice for the link predictor.

**Effectiveness of Two-Phase Training.** By performing our two-phase training, the prediction model achieves a precision of 50.00%, a recall 73.91%, and a $F_1$-score of 59.65%. However, if we directly use 90% of the groundtruth WTGs to train the model, the prediction model achieves a precision of 33.82%, a recall of 67.65%, and a $F_1$-score of 45.10%. This clearly demonstrates the improvement brought by the two-phase training. Moreover, even though the two-phase training improves the performance of the prediction, the overall performance of prediction is still not satisfactory. The reason is that for these unverified edges, ProMal uses static GUI rendering [3, 32] to render the layout files to obtain the view trees and screenshots, which are not as accurate as dynamic analysis. Even so, when it is applied on only the unverified edges, ProMal can achieve further improvement by combining the prediction results of the unverified edges with the verified edges, improving the $F_1$-score to 82.82%. If we use the model without pre-training in ProMal, the precision drops to 61.12%, the recall becomes 78.31%, and the $F_1$-score is only 65.33%, indicating the importance of two-phase training.

---

**Table 4: Details of the dynamic WTGs built by Paladin**

<table>
<thead>
<tr>
<th>App Group</th>
<th># Nodes</th>
<th># Edges</th>
<th>Prec. (Edges)</th>
<th>Recall (Edges)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 1</td>
<td>26</td>
<td>55</td>
<td>100.00%</td>
<td>91.04%</td>
</tr>
<tr>
<td>Group 2</td>
<td>24</td>
<td>44</td>
<td>100.00%</td>
<td>76.36%</td>
</tr>
<tr>
<td>Group 3</td>
<td>26</td>
<td>54</td>
<td>100.00%</td>
<td>65.91%</td>
</tr>
<tr>
<td>Group 4</td>
<td>33</td>
<td>104</td>
<td>100.00%</td>
<td>67.68%</td>
</tr>
<tr>
<td>Group 5</td>
<td>51</td>
<td>122</td>
<td>100.00%</td>
<td>64.31%</td>
</tr>
<tr>
<td>Group 6</td>
<td>45</td>
<td>107</td>
<td>100.00%</td>
<td>46.61%</td>
</tr>
<tr>
<td>Group 7</td>
<td>35</td>
<td>82</td>
<td>100.00%</td>
<td>68.23%</td>
</tr>
<tr>
<td>F-Droid</td>
<td>45</td>
<td>140</td>
<td>100.00%</td>
<td>89.87%</td>
</tr>
<tr>
<td>Total</td>
<td>285</td>
<td>708</td>
<td>100.00%</td>
<td>71.25%</td>
</tr>
</tbody>
</table>

*Prec. and Recall are average values.*
7 DISCUSSION

Static Analysis. ProMal builds the static analysis upon Gator [47, 69, 69], and extends the static analysis to detect dialog transitions buried in the callbacks of apps. While Gator provides a comprehensive model of Android environment (e.g., back stacks, events, and callbacks), it adopts an over-approximation algorithm that applies only weak updates when associating the widget IDs and event handler callbacks. Such problems can be mitigated by applying more expensive but precise analysis such as path-sensitive analysis like FlowDroid [6] and symbolic execution [21, 34, 36, 44, 49].

Dynamic Analysis. Dynamic app exploration [26, 27, 40, 41, 54] has been used to generate GUI tests, discover app behaviors, and detect violations ad fraud. ProMal leverages dynamic app exploration to build the dynamic WTG for an app. While these approaches can automatically trigger both UI widget events and hardware events (e.g., pressing BACK button), they still suffer from coverage issues due to various environment dependencies. This issue can be mitigated by using developer-provided test cases as seeds [8, 71] to improve app exploration.

Window Transition Prediction. Due to the functionality and design differences between different apps, many transition features are quite unique. Therefore, the prediction model has a rather good performance in apps similar to its training set and performs poorly in other apps. In future work, we plan to extract more features such as whether the widget is clickable or not to improve the prediction.

Threats to Validity. Our evaluation subjects may not be representative of the entire market. To mitigate the issue, we choose apps that have WTGs of different complexities, and the subjects used for pre-training (1,625 apps) also alleviate the issue. We plan to address this limitation by including more diversified market apps to further reduce the threats. Also, we discard apps that cannot be analyzed by either PALADIN or GATOR, which are mainly caused by the compatibility problem of the libraries and SDK versions. This can be mitigated by upgrading the libraries and adding support for later SDK versions. Inaccuracies in the manual code inspection are inevitable due to the lack of the ground truth WTGs. In addition, there may be human errors in collecting statistics and studying the evaluation results. These threats are mitigated by double-checking all manual work and ensuring that the results were agreed upon by at least two authors.

8 RELATED WORK

Android UI Modelling. Routev et al.’s GATOR [47, 69, 70] is among the first to provide a static analysis framework for modeling Android apps’ UIs. Gator models GUI-related Android objects, their flow through the application, and their interactions with each other via the abstractions defined by the Android platform [47], and provides a context-sensitive static analysis of callback methods to link callback methods to GUI objects [69]. Built upon this static analysis, Gator further generalizes the analysis with explicit modeling of the window stack to generate a static WTG [70]. Besides static analysis, dynamic analysis builds a UI model that represents different states for windows, which can be considered as a finer grained WTG [26, 40, 41, 54]. These UI states are used to guide the dynamic exploration to either discover more behaviors or identify certain violations (e.g., ad frauds). ProMal’s WTG analysis is built upon these static and dynamic analysis techniques, where the dynamic analysis is used to verify the results of the static analysis.

Hybrid Program Analysis. Hybrid program analysis has been used to improve the precision of various static analysis. Check ‘n’ crash [15, 16] takes the error conditions inferred using theorem proving techniques by a static checker and produces test cases to determine whether an error truly exists. This technique has shown advantages over both static checking and automatic testing individually. Blended analysis [18, 61] combines dynamic and static analysis by first applying dynamic analysis to capture runtime information and then performing static analysis on each dynamic trace to identify solutions, which has been shown to achieve promising results in performance understanding and taint analysis. Another important line of hybrid analysis is concolic testing [21, 49, 57, 66, 67], where dynamic analysis is used to collect constraints along the executed program paths and static symbolic analysis and constraint solver are used to derive new inputs to explore more program paths. ProMal’s “tribrid analysis” is inspired by the idea of hybrid program analysis, where machine learning techniques are used to address the limitations of both static and dynamic analysis.

Machine-Learning Based Link Prediction. Link prediction is a fundamental problem in link mining that attempts to predict the existence of a link between two nodes based on observed links and the attributes of nodes [50]. Link prediction and recommendation problem was first proposed by Liben-nowell et. al. [39]. Traditionally, there are two ways of making link predictions: one way to make this prediction is based on structural properties of the network [39], another way is to make use of attribute information for link prediction [46]. Recently, a lot of methods based on machine learning techniques are proposed [35, 38, 48]. The first to study the link prediction problem as a supervised learning problem is Hasan et. al. [28], where the existing and non-existing social links are treated as the positive and negative instances respectively. ProMal is related to link prediction but focuses on predicting transitions among windows using app-specific UI information.

9 CONCLUSION

We have proposed ProMal, a synergistic analysis that combines static analysis, dynamic analysis, and machine learning to construct a precise WTG. ProMal applies static analysis and dynamic analysis to build the static WTG and the dynamic WTG for an app, and identifies the verified edges by matching the edges in these two WTGs. ProMal then leverages machine learning techniques to predict the unverified edges in the WTG to determine whether they are feasible transitions. Our evaluations on 40 real-world apps demonstrate the superiority of ProMal in building WTGs over static analysis and dynamic analysis when they are applied separately.

ACKNOWLEDGMENTS

Xusheng Xiao’s work is partially supported by the National Science Foundation under the grants CCF-2046953 and CNS-2028748. Xusheng Xiao is the corresponding author.

REFERENCES


