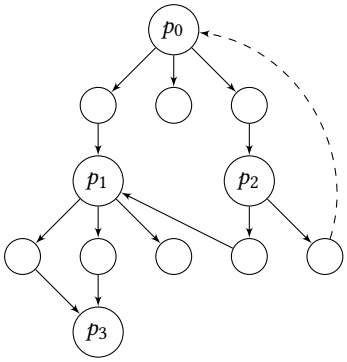

Figure 9: Graph representation of Android UI

- Vertices with labels are V_p, while the others are V_e.

Prioritizing. Each page vertex has outgoing edges linking to element vertices, displaying the relation that a group of elements contained in the same page. When DFS arrives at a page, it is more efficient if some heuristics can be applied to first try those more promising elements rather than choosing randomly. To achieve that, we define a scoring function $F : E_{pe} \mapsto [0, 1]$ to calculate a weight for each of page-to-element edge. More specifically, F is a weighted sum of individual scoring functions for each property of the element the edge points to, namely $F(e) = \sum w_i f_i(e)$. Properties we select as scoring components are *id*, *desc*, *text*, *clickable*, *type*. Weights for each property is set empirically. For instance, *clickable* is a strong indicator comparing to *type*, so we put more weights on it. For individual scoring function, we would like to mention two of our major innovations here.

- *String score function.* We introduce a unified string score function for string type properties (*id*, *desc*, *text*). First, we build a weighted keyword database based on the same keyword statics we used in Algorithm I. Weights are assigned based on the frequency of keywords. Given a string-type property of an element, we search for all substring matches in the keyword database, for each of them we get a raw score s, which is equal to the keyword weight. Instead of directly using this raw score, we calculate an adjusted score $s' = cs$ based on the length of the property string and the matched substring. Finally, the maximum adjusted score among all matches is returned. The adjustment coefficient c is defined by

$$c = \frac{L - x}{Lk^x}, \text{ where } x = \min\{L, l_s - l_m\}$$

In this equation, l_s, l_m represents the length of the property string and matched substring respectively, L is maximum acceptable string length, and k is a parameter controlling the steepness of decrease for the function. We fixed its value to be 1.1. The intuition behind the adjust function is that a string with longer unmatched part usually has less semantic similarity to the matched keyword. For example, a UI widget with text *"login now"* looks more promising as a login button compared to a UI widget with the text *"you can login with email or mobile number"*, even though both of them match the keyword *login*. Here, we set L as an upper bound because widgets with very long text are mostly descriptive, *e.g.*, articles, which is not interactive. Lastly, we did not use linear function (*i.e.*, setting $k = 1$) since the semantic vanishes sharply when the unmatched length increases. Figure 10 shows the curves of the adjust function with different k.

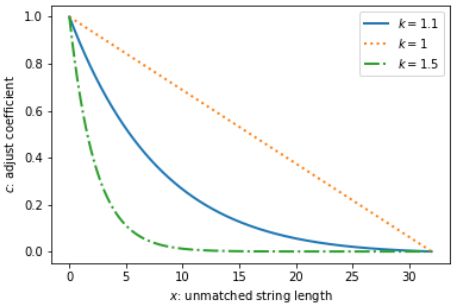

Figure 10: String score adjust function, when $L = 32$

- *Identify clickable elements.* Clickable elements often lead to new pages. Android UI element natively has an attribute called *clickable* which defines whether it reacts to click events. However, elements can overlay on each other and present to users on the same surface. From the user's point of view, an element is clickable if some events can be triggered when tapping on its area, regardless of its *clickable* property. Therefore, we call an element with *clickable* property to be *explicitly clickable*, while element not declared *clickable* but appears to be clickable to users as implicitly clickable. To identify all *implicitly clickable* elements, we calculate the union of areas covered by *explicitly clickable* elements and then mark other elements intersecting with the area to be *implicitly clickable*.

C EXTENSION FOR DETECTING APP SECRET DISCLOSURE

Originally, MoSSOT detected the vulnerability indirectly by monitoring the network traffic through Proxy, where the appearance of critical requests (involving app secret) indicates the vulnerability (Section. 4.3.2). However, the method suffers false negatives as RP developers may hardcode the secret in their APKs without using it. Thus, we extended MoSSOT for better detection accuracy.

Similar to the method in [47], the extended tool forges the critical requests (appended with the app secret) to the IdP server and confirms the leaked secret based on the response content, where the potential app secret is available from the decompilation result of the APK, *i.e.*, Step 1(b) in Fig. 3. Typically, the kind of requests involves three parameters, namely app id, app secret, and a credential (*e.g.*, access token), and the IdP server verifies them in the same order. Thus, once the app id is correct, we can get the judgment of the chosen app secret from the IdP server. Previous work [47] attempted to obtain the app id via reverse engineering and might fail due to the protection on the APK. In contrast, our tool extracts the app id from the reference network traffic (Step.4 in Fig. 3), which always appears in the interactions between the IdP app and its server (*e.g.*, Step 3 in Fig. 1). Therefore, our tool is expected to have higher accuracy in detecting the vulnerability.

With the extension, MoSSOT detected 187 vulnerable apps and 99 of them were not captured by the original method.