UI Driven Android Application Reduction

Jianjun Huang[1], Yousra Aafer[1], David Perry[1], Xiangyu Zhang[1], Chen Tian[2]
1. Department of Computer Science, Purdue University, USA 2. Huawei R&D, USA
{huang427, yaafer, perry74}@purdue.edu, xyzhang@cs.purdue.edu, chen.tian@huawei.com

***Abstract*—While smartphones and mobile apps have been an integral part of our life, modern mobile apps tend to contain a lot of rarely used functionalities. For example, applications contain advertisements and offer extra features such as recommended news stories in weather apps. While these functionalities are not essential to an app, they nonetheless consume power, CPU cycles and bandwidth. In this paper, we design a UI driven approach that allows customizing an Android app by removing its unwanted functionalities. In particular, our technique displays the UI and allows the user to select elements denoting functionalities that she wants to remove. Using this information, our technique automatically removes all the code elements related to the selected functionalities, including all the relevant background tasks. The underlying analysis is a type system, in which each code element is tagged with a type indicating if it should be removed. From the UI hints, our technique infers types for all other code elements and reduces the app accordingly. We implement a prototype and evaluate it on 10 real-world Android apps. The results show that our approach can accurately discover the removable code elements and lead to substantial resource savings in the reduced apps.**

I. Introduction

Smartphone apps have become an integral part of our daily life [1]. However, apps tend to contain a lot of rarely used functionalities. For example, advertisements are reported to appear in more than 50% of Android apps [2]. As ad libraries are provided by third party service providers and integrated by developers into their apps, this practice has raised lots of privacy and security concerns [3]. In fact, previous studies have identified sensitive information exposure in ad networks [4], [5], and others have found that upon a user's click, advertisements may reach some destinations that play an important role in propagating attacks [6]. In addition to advertisements, apps may offer extra features that are usually not desired by the users. For example, a weather app may recommend news stories, and a calendar app may include news-like sections (as we will show in Section IV-B2).

Besides the non-relevant features, some apps may contain complex functionalities that are relevant to the apps' purpose, but considered redundant or distracting by the users. For example, a shopping app usually recommends products based on the user's profile and shopping history. However, uninterested users might find such recommendations quite distracting, and would appreciate it if the app provides an option to turn this feature off. Besides visual distraction, unwanted functionalities often incur additional consumption of battery power, CPU cycles, bandwidth and so on. Previous studies have shown that mobile apps using ads consume significantly more network data and have increased energy consumption [7].

Therefore, there is an increasing need of *automatically* customizing mobile apps to meet the various demands of different user groups. For example, enterprises and government agencies may want the apps installed on their employee's devices to not have potentially malicious third party components (*e.g.,* ad components). Users that often operate their devices in rough environments such as outdoors and battle fields may want to minimize battery and data consumption by turning off unnecessary app features. Even normal users may have different personal preferences/needs to customize apps. For example, parents may want to disable components that could deliver inappropriate content to their children. Unfortunately, due to the potentially diverse needs, app customization and personalization are prohibitively expensive in terms of human effort for the development team if no automatic tools are available.

In this paper, we propose a UI driven app customization technique that removes unwanted features in Android apps that are associated with given UI elements. In particular, our technique requires the user to first specify the UI elements denoting functionalities that she wants to remove. Using this information, our technique automatically identifies the program locations that load the specific UI elements into code and store them in variables. With such program locations, our technique can then track all correlated uses of the UI elements and the code elements for relevant background tasks, *e.g.,* acquiring data from remote servers. While the UI related code elements are directly removable as indicated by the user, the background tasks require more analysis to determine if they are removable. Specifically, our technique examines whether the background tasks generate data that exclusively flows to the specified UI elements. If so, these background tasks are considered removable. Otherwise (*i.e.,* the generated data gets propagated to other components), the background tasks cannot be removed. The underlying analysis is a type system, in which each code element is tagged with a type indicating if it should be removed. The type system infers types for all correlated code elements from the UI hints. Finally, our technique reduces the app according to the deduced types.

Our technique can be potentially used in the last stage of app development to generate a large number of customizations to meet different needs. In addition, it is designed in such a way that it does not require source code. As such, legacy apps whose source code and original developers are no longer available can benefit from our technique as well.

We evaluate our prototype on 10 popular Android apps. Our evaluation shows a substantial reduction of various resources

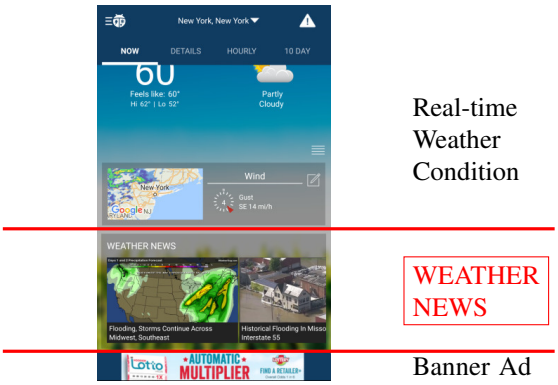

Fig. 1. Screenshot of app WeatherBug.

usage. On average, removing certain UI elements associated with typical unwanted functionalities results in saving 28.1% of data usage. 34.1% of CPU time, 26.2% Wi-Fi running time, and 37.6% of computed power use. Our achieved results clearly demonstrate the strength of our proposed UI driven application reduction.

Our work makes the following contributions:

- We propose a UI driven approach to remove unwanted functionalities in Android apps associated with UI elements specified by the user. Our technique features a type system that tags all relevant code elements to indicate if they are removable.
- We implement a prototype and evaluate it on a set of real-world Android apps. The results show that our approach accurately discovers the removable code elements and the reduction leads to substantial resource savings.

II. Motivation

We use WeatherBug [8], a real-world Android app to motivate our technique. WeatherBug is a popular app that provides weather alerts, real-time weather conditions, hourly forecasts and much more. While the status bar notification provided by WeatherBug is enough to check live weather conditions, the user can launch the app to receive more detailed information and access more advanced features. Intuitively, once the app is launched, the user expects to get a view of real-time weather conditions. However, the main page offers much more additional information. As shown in Figure 1, the main page of the app contains the real-time weather condition, some weather news and a banner ad. If we further scroll down, the page contains more unrelated information such as photos from other users, the closest spark strike and so on.

Besides overwhelming the user with unrelated information, these components consume additional energy and network data. Thus, there is a need to customize/refactor the app to remove some features that are not essential to various clients. For example, to prevent visual distraction, reduce potential privacy leaks and malicious behaviors (in an enterprise environment) [9], [3], [6], and to reduce energy and network data consumption (in rough environments such as outdoors). *The overarching idea of our work is to customize an app to meet different user needs by specifying what features are not needed on the UI.*

Suppose the user wants to remove some unwanted UI components and any associated functionalities in the WeatherBug app. Specifically, consider the case in which the user selects `WEATHER NEWS`, the highlighted component in Figure 1 for removal. Figure 2 shows simplified code snippets representing the corresponding work flow where ⓝ describes the execution order of a method. When the component is loaded, the Android OS invokes ① `onCreateView` which inflates a static layout file at line 4 to hold the content of the component. Additionally, a `StoriesAdapter` is created and added to the component. This provides a binding from the component-specific data set to views that are displayed, *e.g.,* the images and the titles of all inner elements. The Android OS then invokes ② `onActivityCreated` that eventually launches a background functionality for acquiring data from a remote server. Once this data is loaded, the background task invokes its callback method ③ `onRequestCompleted`. In this callback method, the data is transmitted to a thread and the thread is queued for execution. Later, the Android OS executes ④ `run` in the thread. In `run`, the data is retrieved and saved to a shared data set at line 30. Next, the thread notifies `StoriesAdapter` about the data set change at line 31. Finally, this data set change notification triggers the execution of ⑤ `onCreateViewHolder` and ⑥ `onBindViewHolder`. The method `onCreateViewHolder` creates a view holder with an inflated view, and the method `onBindViewHolder` obtains data from the shared data set and displays it on the inflated view at line 45.

In order to remove the unwanted component, *i.e.,* the UI elements and functionalities associated with `WEATHER NEWS`, our approach first receives the information associated with the elements specified by the user. Next, it uses this information to discover the program locations loading the UI elements into code. Then it takes four steps to remove the UI elements and associated functionalities: 1) forwardly discovers all code related to the specified UI elements; 2) starting from where the UI elements use data, backwardly tracks the data to its generation points in background functionalities; 3) examines whether the data obtained from the discovered data generation points are exclusively used by the specific UI elements or not; 4) iteratively removes the code.

In our motivating example, lines 4, 37 and 43 are associated with the unwanted UI elements. Our technique forwardly tracks the uses of the correlated variables, as depicted by → in Figure 2 and identifies lines 7 and 45 that put data to the specific UI elements. Line 7 binds `StoriesAdapter` to the UI component to respond to data changes and displays the content, acting similarly to an action handler. We omit the discussion of tracking the adapter for simplicity. Line 45 displays a short description of the corresponding image as shown in Figure 1 using the given data. Starting from line 45, we backwardly track the data along ⇢ `onRequestCompleted` and then the data generation point in the background functionality (omitted in the code). We further find the location

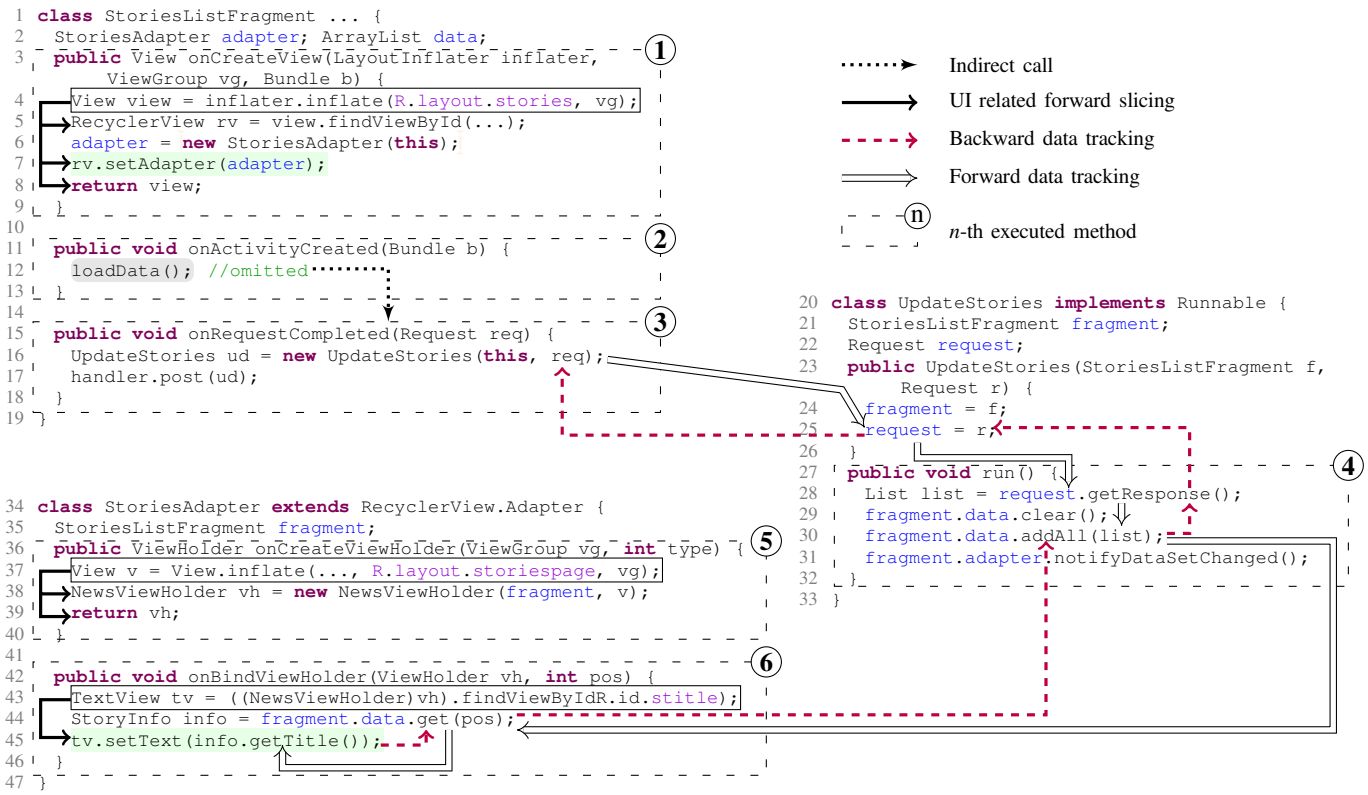

Fig. 2. Simplified code snippet for the WEATHER NEWS component and the corresponding flows.

Fig. 3. Removing WEATHER NEWS.

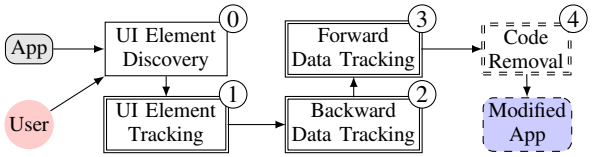

Fig. 4. Approach overview.

invoking the background functionality at line 12. Next, we check whether the generated data is only used in the specific UI elements by forwardly tracking the data from the data generation point following ⇒ in Figure 2. We do not find any other components using the data in the example and thus the data is specific to the UI elements. However, the background functionality is found to be a public component used at other locations in the app and thus the discovered code inside the functionality, including the data generation point, cannot be removed. Finally, our technique removes the other code not occurring in the background functionality and iteratively removes methods and classes if applicable.

To show the effect of the code reduction, our technique replaces the layout inflation in `onCreateView` with a dummy view and eliminates all the other removable code elements. The modified code snippet and the runtime screenshot is shown in Figure 3. The dummy view is depicted in the highlighted area. As shown in Section IV, removing the `WEATHER NEWS` component reduces 9% of the data usage and 20% of the power consumption.

III. Design

In this section, we will discuss our UI driven approach to discovering removable code elements in Android apps. Figure 4 shows the work flow of our approach. As discussed in Section II, after receiving the user specified UI elements and discovering the program locations referring to them (step ⓪), our approach takes four steps to detect potentially removable code elements: ① forwardly discovers all uses of the specified UI elements; ② backwardly tracks the data used in the UI elements;③ finds all uses of the data, starting from the discovered data generation points; ④ iteratively removes the code elements based on the results of data tracking. Steps ⓪ and ① are discussed in Section III-C, Section III-D describes steps ② and ③, and Section III-E talks about step ④.

We use the code snippet in Figure 5 throughout the section to exemplify the approach details. Assume that `id0` and `id1` refer to unwanted UI elements but `id2` doesn't. `BgTask` represents a background functionality that obtains input data from remote servers and stores the result to `data`. Line 13 abstracts the operations of obtaining input data by function `getInputData`. A concrete example is `HttpClient.execute()` which is commonly used in Android

```
1  // class A declares field "data"
2  class B extends A {
3    a() { t = new BgTask(this);trigger(t);  }
4    b() { textView0 = findViewById(id0);//unwanted
5          textView0.setText(data);  } }//``B''
6  class C extends A {
7    c() { g = new BgTask(this);trigger(g);  }
8    d() { textView1 = findViewById(id1);//unwanted
9          textView1.setText(data.split(":")[0]);
10         textView2 = findViewById(id2);//not unwanted
11         textView2.setText(data.split(":")[1]); } }//``C''
12 class BgTask { A ck; BgTask(A a) {ck = a;}
13   void run() { ck.data = getInputData(); }}//``BgTask''
```

Fig. 5. Code example for discussion.

Program	P	::=	K*	
Class	K	::=	M*	
Method	M	::=	m(x) {s*}	
Statement	s	::=	$x :=^{l}$ c	/*constant*/
		|	$x :=^{l}$ inflate(i)	/*inflation*/
		|	$x :=^{l}$ findViewById(i)	/*get view*/
		|	$x :=^{l} \ominus y$	/*unary assignment*/
		|	$x :=^{l} y \oplus z$	/*binary assignment*/
		|	$x :=^{l} \phi(y, z)$	/*value merging in SSA*/
		|	$x :=^{l}$ y.z	/*get field*/
		|	$x.y :=^{l} z$	/*put field*/
		|	$x :=^{l}$ checkcast(y)	/*type cast*/
		|	$x :=^{l}$ new k(y)	/*new instance*/
		|	$x :=^{l}$ y.m(z)	/*method call*/
		|	$return^{l}$(x)	/*return in a method*/
Variable	x, y, z			/*all variables*/
ID	i			/*UI-related Id*/
Value	c			/*non-Id constant*/
Label	l			/*{l_1, l_2, l_3, ...}*/

Fig. 6. Simplified language model.

apps to fetch remote data from Web servers. The data acquisition task is initiated at two locations: line 3 and line 7. The data is displayed on UI elements at lines 5, 9 and 11.

A. Language Abstraction

To simplify our discussion, we introduce an abstract language, as presented in Figure 6. A program is made up of classes, a class contains a list of methods, and each method contains a number of statements. We model common types of statements and other operations are abstracted away or simplified. We label statements with a superscript.

As we discussed in Section II, we start our analysis from the layout inflations and aim to remove the inflated views. Thus, we introduce the function *inflate* to represent all kinds of inflations in the code (*e.g.,* lines 4 and 37 in Figure 2). Additionally, we introduce another function *findViewById* to represent the operation of an app looking for a UI element.

We abstract all types of assignment, such as unary assignment, binary assignment, type cast assignment and assignment from/to a field variable. Note that our language is a kind of single static assignment (SSA) language such that conditionals (including loops) are implicitly represented by the value merge statement $\phi(y, z)$. Since the predicate of the value merge statement is irrelevant in our analysis, it is abstracted away and hence y and z denote the values of the same variable in the two branches of a conditional.

[s]	:=	statementsIn(m)	[m]	:=	methodsIn(k)
x	:=	thisOf(k)	x	:=	paramOf(m)
s	:=	returnOf(m)			

Fig. 7. Functions for *class*, *method* and *statement*.

For object creation (*i.e.,* new instance statement) and method calls, we assume there is only one parameter besides the receiver object (‘this’) of an instance method call. A return statement returns a value from a callee method to a variable in the caller method.

We also define a number of auxiliary functions for statements, methods, and classes to acquire correlated information during analysis. These functions are shown in Figure 7. Function *statementsIn* returns all statements inside a method, represented by [s]. Similarly, *methodsIn* returns all declared methods in a class k. Functions *thisOf* and *paramOf* behave similarly, except that one looks for ‘this’ reference in the callee method and the other searches the corresponding formal parameter. The return statement of a method is given by *returnOf*.

B. Type System

We formalize our approach in a type system. Code elements, including variables, statements, methods, and classes are associated with tags. A tag is treated as the type T of the code element l. We say l has type T, written as $l : T$. The type of a code element may be a set of tags, for example, $l : \{T, T'\}$. In this case, We use “$\cup$” to union two sets of types together. In this paper, we define the type domain as:

$$Type = \left\{ \begin{array}{lll} \texttt{UIRelated} & \texttt{UIData} & \texttt{InputUIData} \\ \texttt{Removable} & \texttt{Unremovable} & \end{array} \right\}$$

Type UIRelated is used to mark the variables or statements that depend on the specified UI elements. We type the code elements with it and propagate it at step ①. For example, we type an inflate() method call where the inflated view is a user specified UI element and the resultant variable with UIRelated. When we find any data uses on the specified UI elements, we type the data variable with UIData and then backwardly propagate it at step ②. If we reach a data generation point along above propagation, we get the knowledge that the data is some input data from outside sources. We type the data generation point and the data variable with InputUIData and propagate it forwardly (step ③). Eventually, we can mark which code elements are removable or definitely unremovable using the corresponding types (step ④).

Notice that type InputUIData can be parameterized with a data generation point. It will help us distinguish the types of code elements which are correlated with multiple data generation points.

In addition to the concrete types, if any code element has not been visited or is not interesting to us, we say it is not typed, in the notation of *nil*.

Consider that the same code element executed in different calling contexts may produce different results, and in some contexts it may be related to the specified UI elements while

$$\text{UI-Inflate}\ \frac{belongToSpecifiedUI(i)}{\Gamma_1, \{x :=^l inflate(i)\}_\epsilon \models \Gamma_1 \Rightarrow [x_\epsilon, l_\epsilon : \texttt{UIRelated}]\Gamma_1}$$

$$\text{UI-FindView}\ \frac{belongToSpecifiedUI(i)}{\Gamma_1, \{x :=^l findViewById(i)\}_\epsilon \models \Gamma_1 \Rightarrow [x_\epsilon, l_\epsilon : \texttt{UIRelated}]\Gamma_1}$$

$$\text{UI-Call-This}\ \frac{nonApi(m) \quad \Gamma_1 \vdash y_\epsilon : \texttt{UIRelated} \quad \epsilon' = \epsilon \cdot l \quad t = thisOf(m)}{\Gamma_1, \{x :=^l y.m(z)\}_\epsilon \models \Gamma_1 \Rightarrow [x_\epsilon, t_{\epsilon'}, l_\epsilon : \texttt{UIRelated}]\Gamma_1}$$

$$\text{UI-BAssign}\ \frac{\Gamma_1 \vdash y_\epsilon : \texttt{UIRelated}}{\Gamma_1, \{x_\epsilon :=^l y_\epsilon \oplus z_\epsilon\}_\epsilon \models \Gamma_1 \Rightarrow [x_\epsilon, l_\epsilon : \texttt{UIRelated}]\Gamma_1}$$

$$\text{UI-Return}\ \frac{\Gamma_1 \vdash r_{\epsilon'} : \texttt{UIRelated} \quad \epsilon' = \epsilon \cdot l \quad x :=^l y.m(z)}{\Gamma_1, \{return^{l'}(r)\}_{\epsilon'} \models \Gamma_1 \Rightarrow [x_\epsilon, l'_{\epsilon'}, l_\epsilon : \texttt{UIRelated}]\Gamma_1}$$

Fig. 8. Rules for discovering code elements associated with the UI elements.

in other contexts it may not be. We use ϵ to represent the calling context which is a stack of labels referring to method calls or new instance sites. Each code element l is tagged with a context (*e.g.,* l_ϵ) in order to conduct context-sensitive analysis. Different contexts are depicted by the subscript of ϵ, *e.g.,* ϵ_1 and ϵ_2. We use "$\cdot$" to concatenate a context with a label to form a new context for the statements in the callee method associated with l, for example, $\epsilon' := \epsilon \cdot l$.

The mappings from the code elements to the types form the context Γ of the type system, which is iteratively updated during analysis until a fixed point is reached. For example, at the beginning, Γ is empty. Upon a removable statement l_ϵ, Γ is updated to $\{l_\epsilon$: `Removable`$\}$. At this point, we have $\Gamma \vdash l_\epsilon$: `Removable`, which means under (type) context Γ, statement l_ϵ is typed with `Removable`. In other words, $\Gamma(l_\epsilon)$ = `Removable`, where $\Gamma(l_\epsilon)$ evaluates statement l_ϵ in the context to obtain the corresponding type.

When a statement l_ϵ is evaluated, the context may be updated. We use $\Gamma, l_\epsilon \models \Gamma \Rightarrow \Gamma'$ to indicate that under type context Γ, evaluating code l_ϵ updates the context from Γ to Γ'. We use $[l_\epsilon : T]\Gamma$ to represent an update to the context. Specifically, if no mapping is found for l_ϵ in Γ, the mapping is added to the context. But if there exists a mapping for l_ϵ, the rule substitutes the existing type of l_ϵ with type T. Multiple mappings can be updated simultaneously. For instance, $[l_{\epsilon_1} : T, l'_{\epsilon_2} : T']\Gamma$ update the context for two code elements l_{ϵ_1} and l_{ϵ_2}. We also use $[l_{\epsilon_1}, l'_{\epsilon_2} : T]\Gamma$ to denote $[l_{\epsilon_1} : T, l'_{\epsilon_2} : T]\Gamma$ for brevity.

In the following analysis, we define four type contexts: Γ_1, Γ_2, Γ_3 and Γ_4 for the four steps respectively. We also have a special context MR in which the discovered data generation points are mapped to *True* or *False*, indicating whether the corresponding data generation points must be retained or not.

C. Turning Off UI Element

In this section, we discuss how to identify the code elements related to the unwanted UI elements, *i.e.,* steps ⓪ and ① in Figure 4. Starting from *inflate* and *findViewById*, we forwardly track the uses of the specified UI elements and type all correlated statements and variables with `UIRelated`.

We define the rules in Figure 8. We omit the rules for some statements, *e.g.,* unary assignment, ϕ assignment and field access, due to the space limit. In addition, we do not present the rules for API method calls that require models. New instance operations behave similarly to method calls and thus we omit the corresponding rules too.

Based on the language definition, the user specified UI elements are introduced into the code through layout inflation (*inflate*) or view finding (*findViewById*). We apply rules UI-Inflate and UI-FindView to start the analysis. If the given id is corresponding to a specified UI element, we say the resultant variable and the statement are related to specific UI element.

If the receiver object `y` of a method call `m` is typed with `UIRelated`, the definition of `y` will be *potentially* removed. The removal results in a null reference, leading to run-time exceptions. Therefore, we apply rule UI-Call-This in this case to type the resultant variable `x`, the method call statement l and the corresponding '`this`' reference in the callee method with `UIRelated`. If an actual argument `z` is typed, the behavior is similar to UI-Call-This and the corresponding rule is omitted. Rule UI-BAssign indicates that if a right-hand-side variable is typed, the left-hand-side variable and the statement are typed too. Rule UI-Return is applied when a return variable in a callee method is typed with `UIRelated`. It propagates the type to the resultant variable at the corresponding call site and types the call site with `UIRelated`.

Example: Consider applying the rules to the code snippet in Figure 5. First, we type `textView0` and `textView1`, lines 4 and 8 with `UIRelated` by rule UI-FindView. Then through type propagation, lines 5 and 9 are typed with `UIRelated`. We present the context updates as follows. ϵ_1 is the calling context of method `B.b()` and ϵ_2 is the calling context of method `C.d()`.

$$\text{UI-FindView(Line } 4_{\epsilon_1}\text{, Line } 8_{\epsilon_2}) \Rightarrow \begin{bmatrix} \text{textView0}_{\epsilon_1} & : & \texttt{UIRelated} \\ \text{textView1}_{\epsilon_2} & : & \texttt{UIRelated} \\ \text{Lines } 4_{\epsilon_1}, 8_{\epsilon_2} & : & \texttt{UIRelated} \end{bmatrix} \Gamma_1$$

$$\text{UI-Call-This(Line } 5_{\epsilon_1}\text{, Line } 9_{\epsilon_2}) \Rightarrow \left[\ \text{Lines } 5_{\epsilon_1}, 9_{\epsilon_2}\ :\ \texttt{UIRelated}\ \right] \Gamma_1$$

D. Discovering Associated Background Functionalities

If we reach some API method calls that put data to the specified UI elements for display, we track where the data is generated. We *backwardly* track the generation of the data and type all the involved statements and associated variables with `UIData` (step ②). When we reach a data generation point that acquires data from outside sources (*e.g.,*`HttpClient.execute()`), we use `InputUIData`

$$\text{UI-Put-Data}\ \frac{\Gamma_1 \vdash y_\epsilon : \texttt{UIRelated} \quad \Gamma_1 \vdash l_\epsilon : \texttt{UIRelated} \quad apiPutDataToUI(m)}{\Gamma_1, \{x :=^l y.m(z)\}_\epsilon \models \Gamma_2 \Rightarrow [z_\epsilon : \texttt{UIData}]\Gamma_2}$$

$$\text{Bwd-Assign}\ \frac{\Gamma_2 \vdash x_\epsilon : \texttt{UIData}}{\Gamma_2, \{x :=^l y \oplus z\}_\epsilon \models \Gamma_2 \Rightarrow [y_\epsilon, z_\epsilon, l_\epsilon : \texttt{UIData}]\Gamma_2}$$

$$\text{Bwd-Call-Return}\ \frac{\Gamma_2 \vdash x_\epsilon : \texttt{UIData} \quad nonApi(m) \quad \epsilon' = \epsilon \cdot l \quad \{return^{l'}(r)\} = returnOf(m)}{\Gamma_2, \{x :=^l y.m(z)\}_\epsilon \models \Gamma_2 \Rightarrow [r_{\epsilon'}, l'_{\epsilon'}, l_\epsilon : \texttt{UIData}]\Gamma_2}$$

$$\text{Bwd-Call-Param}\ \frac{nonConstructor(m) \quad \epsilon' = \epsilon \cdot l \quad p = paramOf(m) \quad \Gamma_2 \vdash p_{\epsilon'} : \texttt{UIData}}{\Gamma_2, \{x :=^l y.m(a)\}_\epsilon \models \Gamma_2 \Rightarrow [a_\epsilon : \texttt{UIData}]\Gamma_2}$$

Fig. 9. Rules for backwardly discovering data relevant code elements.

$$\text{Bwd-Call-Data-Gen}\ \frac{\Gamma_2 \vdash x_\epsilon : \texttt{UIData} \quad apiGetInputData(m)}{\Gamma_2, \{x :=^l y.m(z)\}_\epsilon \models \Gamma_3 \Rightarrow [x_\epsilon, l_\epsilon : \Gamma_3(l_\epsilon) \cup \{\texttt{InputUIData}(l)\}]\Gamma_3}$$

$$\text{Fwd-Assign}\ \frac{\texttt{InputUIData}(l^d) \in \Gamma_3(y_\epsilon)}{\Gamma_3, \{x :=^l y \oplus z\}_\epsilon \models \Gamma_3 \Rightarrow [x_\epsilon, l_\epsilon : \Gamma_3(x_\epsilon) \cup \{\texttt{InputUIData}(l^d)\}]\Gamma_3}$$

$$\text{Fwd-Call-This}\ \frac{nonApi(m) \quad \texttt{InputUIData}(l^d) \in \Gamma_3(y_\epsilon) \quad \epsilon' = \epsilon \cdot l \quad t = thisOf(m)}{\Gamma_3, \{x :=^l y.m(z)\}_\epsilon \models \Gamma_3 \Rightarrow [x_\epsilon, t_{\epsilon'}, l_\epsilon : \Gamma_3(x_\epsilon) \cup \{\texttt{InputUIData}(l^d)\}]\Gamma_3}$$

$$\text{Fwd-Call-Return}\ \frac{\texttt{InputUIData}(l^d) \in \Gamma_3(r_{\epsilon'}) \quad \epsilon' = \epsilon \cdot l \quad x :=^l y.m(z)}{\Gamma_3, \{return^{l'}(r_{\epsilon'})\}_{\epsilon'} \models \Gamma_3 \Rightarrow [x_\epsilon, l_\epsilon, l'_{\epsilon'} : \Gamma_3(x_\epsilon) \cup \{\texttt{InputUIData}(l^d)\}]\Gamma_3}$$

$$\textbf{Unexpected-Data-Use}\ \frac{\texttt{InputUIData}(l^d) \in \Gamma_3(z_\epsilon) \quad \texttt{UIRelated} \notin \Gamma_1(y_\epsilon) \quad apiPutDataToUI(m)}{\Gamma_1, \Gamma_3, \{x :=^l y.m(z)\}_\epsilon \models MR(l^d) \rightarrow True}$$

Fig. 10. Rules for discovering the uses of input data.

to type the data variable and the statement, indicating their correlation with input data. We then *forwardly* propagate `InputUIData` along data flows (step ③). Along the forward propagation, we parameterize `InputUIData` with l^d, a data generation point, to distinguish data originating from different points. If we encounter any cases in which the tracked data is used in some UI components other than the unwanted ones, we need to remember that the corresponding data generation points from which the data propagations are unremovable. We define the backward propagation rules in Figure 9 and the forward rules in Figure 10.

Step ② starts with the discovery of API calls putting data on specified UI elements. Rule UI-Put-Data initiates the context Γ_2 which stores mappings from variables or statements to type `UIData`, by typing the data variable with `UIData`. Bwd-* rules are then applied to propagate the type backwardly. If a resultant variable `x` is typed in a method call, the corresponding return value `r` in the callee method should also be typed (Bwd-Call-Return). If a formal parameter `p` in a callee method is typed, the corresponding actual argument at the caller is typed (Bwd-Call-Param). If the typed variable is `this` reference in the callee method, we type the corresponding receiver object.

During the propagation, if we meet a API method call that acquires outside data and stores the data to any variables under tracking, we consider the method call as a data generation point and thus type the data variable and the data generation point with `InputUIData` in context Γ_3 (Bwd-Call-Data-Gen). Given the fact that a code element may be correlated with multiple data generations, we associate each type `InputUIData` with the location of data generation and we use a set to represent the type of a code element. For example, statement `a = b · c` concatenates two strings and `b` is generated by a method call at l^0 while `c` is generated at l^1. Therefore, `a` has a type of {`InputUIData`(l^0, l^1)}. If either one of the data generation points cannot be removed, this statement is unremovable. The forward propagation rules are straightforward, similar to the ones for discovering UI related code elements, except that type `InputUIData` is parameterized with a data generation point and the resultant type is a union of the incoming type and the original types.

If the input data is discovered to be used by some UI elements that are not correlated with the unwanted UI elements, we apply rule Unexpected-Data-Use and mark in context MR that the corresponding data generation point must be retained, *i.e.,* unremovable.

Example: We apply the rules to the example in Figure 5. Backward data tracking updates Γ_2 as:

$$[\ \text{data}_{\{...;3\}}, \text{data}_{\{...;7\}}, \text{Line } 13_{\{...;3\}}, \text{Line } 13_{\{...;7\}}\ :\ \texttt{UIData}\ ]\ \Gamma_2$$

We use {...;n} to denote the calling context of method `run()` where *n* denotes the line number of corresponding trigger site. The background functionality is triggered in two contexts, one at line 3 and the other at line 7. This results in different instances of variable `data` and the statement of getting input data. Therefore, variable `data` at line 13 is defined in two calling contexts, represented by $data_{\{...;3\}}$ and $data_{\{...;7\}}$. Starting from the data generation points, we have

$$\left[\begin{array}{r} \text{data}_{\{...;3\}}, \text{Line } 13_{\{...;3\}}, \text{Line } 5_{\epsilon_1}\ :\ \{\texttt{InputUIData}(\text{Line } 13)\} \\ \text{data}_{\{...;7\}}, \text{Line } 13_{\{...;7\}}\ :\ \{\texttt{InputUIData}(\text{Line } 13)\} \\ \text{Line } 9_{\epsilon_2}, , \text{Line } 11_{\epsilon_2}\ :\ \{\texttt{InputUIData}(\text{Line } 13)\} \end{array}\right] \Gamma_3$$

At line 11, the data is sent to a unspecified UI element and the line is typed with `InputUIData`(Line 13). With

rule Unexpected-Data-Use, we mark the corresponding data generation point as $MR(\text{Line } 13) \rightarrow True$, saying that the corresponding data generation point must be retained.

E. Removing Code Elements

After we type all data correlated statements with one or more `InputUIData` types, we can finally determine which code elements are removable or unremovable (step ④). The rules are shown in Figure 11. We use l to aggregate the code element in *all* calling contexts, *i.e.,* l_{ϵ_j} for all j.

Rule Remove-Stmt-1 says, if a code element l is directly related to the specified UI elements in all possible calling contexts, it is absolutely removable. If l in a context ϵ has a type of `InputUIData` which is associated with some data generation point l^d, and the data generation point needs to be retained when l is not a UI related code element, rule Unremove-Stmt declares l as unremovable. In contrast, if the corresponding data generation point is not required to be retained and the target l has not been typed with `Unremovable`, we use Remove-Stmt-2 to mark l as removable. Rules Remove-Method and Remove-Class behave similarly. If all call sites of a method or instantiation sites of a class are removed, the method or the class can be fully removed. The last two rules iteratively remove included code elements when a method or a class is removable.

We are also able to find out the trigger sites of background functionalities, like the `start()` call of a `Thread` object or `execute()` call on an `AsyncTask` instance. If the triggered operations in background functionalities are all removable, we can remove the corresponding trigger sites as well. Furthermore, we do not remove branch statements like `if` and `switch` even if they use variables whose definitions are removable. Instead, we replace the definition statement of each such variable with a statement that assigns 0 or `null` to that variable, depending on the type of the variable.

Example: We can now apply the rules to remove code elements in Figure 5. We have

$$\left[\begin{array}{r} \text{Line 4, Line 5, Line 8, Line 9 : } \texttt{Removable} \\ \text{Line 11, Line 13 : } \texttt{Unremovable} \end{array}\right] \Gamma_4$$

Therefore we can remove lines 4, 5, 8 and 9. The background data generation point at line 13 cannot be removed because its data flows to non-specified UI elements at line 11. However, under the context of class `B` where the data is only used in the unwanted UI elements, we can disable the corresponding trigger of the background functionality to avoid unnecessary network access after we remove the unwanted UI elements. We discover the corresponding trigger site at line 3 and we type it with `Removable` while we retain the trigger site at line 7.

IV. Evaluation

We implement a prototype TOFU, turning off UI elements, to discover the removable code elements for specified unwanted UI elements. TOFU is built on top of Soot [10], supporting rewriting modified code to DEX files. We evaluate TOFU on 10 popular Android apps (Table I). TOFU runs on an Intel Core i5 2.5GHz machine with Windows 10 and 8GB memory.

TABLE I
BENCHMARK APPS AND REMOVED UI.

App	Removed UI	Description
WeatherBug	WEATHER NEWS	Weather related news
Dictionary.com	Blog/Slideshow	Word related items on main page
Baidu iKnow	Latest Q&A	Real-time update on main page
Walmart	Recommendation	Recommended items based on the other users' choices
Macy's		
China Calendar	News Button	A button to news page
Fox News	Sponsored Stories	Ad in each news page
CBS News	Banner Ad	Banner ad associated with each news title and brief introduction
AP Mobile	Banner Ad	Bottom banner ad in each page
Tattoo My Photo	Banner Ad	

In our experiments, we specify the UI elements that represent the unwanted features based on our understanding of the expected app behavior. The unwanted components include normal functionalities that certain users may not need (*e.g.,* word related additional information on the main page in a dictionary app), irrelevant buttons (*e.g.,* a button leading to a news page in a calendar app) and advertisements. More details can be found in Table I. We manually obtain the corresponding IDs of the specified UI elements. TOFU then automates the aforementioned analysis and the generation of the modified APK file. Specifically, given an app, we run it to the pages containing UI components correlated with the unwanted functionalities in a Nexus 6P running Android 6.0.1. We then use the Android SDK tool UIAutomator Viewer to obtain the dynamic UI hierarchy, on which we select the components to be removed (corresponding to the unwanted features). We feed the corresponding information (IDs of layouts and UI elements) to TOFU. TOFU then statically analyzes the app code, removes the code elements marked as removable and rewrites the modified code to DEX files, constructing a new APK file. Next, we sign the APK file and run it in the same device in order to measure the savings resulting from our code reduction.

Each pair of apps (original and modified) are kept running for 10 minutes and then their power use is examined. Since the power consumption profiler rounds the result up to an integer, it may not be able to tell the differences in some cases. If the values are not distinguishable, we continue to test the pair of apps for another 20 minutes. For each app, at the beginning of first run, we disable potential live content update (*e.g.,* push notifications) that may largely influence the results. For example, the app AP Mobile updates its displayed live news in the background. If we allow all the categories (*e.g.,* Sports, Entertainment) to update, in a 10-minute run, we observed 73MB of data usage, 9 minutes of CPU time and 20 mAh of power use. After unsubscribing all categories except the Top News, the data usage is reduced to less than 10MB.

A. Experiment Results

Our evaluation aims to measure the benefits of our code reduction with regards to two aspects: data usage and battery usage. To evaluate the data usage reduction, we select

$$\text{Remove-Stmt-1}\ \frac{\Gamma_1 \vdash l : \texttt{UIRelated}}{\Gamma_4 \Rightarrow [l : \texttt{Removable}]\Gamma_4}$$

$$\text{Unremove-Stmt}\ \frac{\texttt{InputUIData}(l^d) \in \Gamma_3(l_\epsilon) \quad MR(l^d) \quad \Gamma_1(l) \neq \texttt{UIRelated}}{\Gamma_4 \Rightarrow [l : \texttt{Unremovable}]\Gamma_4}$$

$$\text{Remove-Stmt-2}\ \frac{\texttt{InputUIData}(l^d) \in \Gamma_3(l_\epsilon) \quad \neg MR(l^d) \quad \Gamma_4(l) \neq \texttt{Unremovable}}{\Gamma_4 \Rightarrow [l : \texttt{Removable}]\Gamma_4}$$

$$\text{Remove-Method}\ \frac{nonApi(m) \quad \forall(x :=^l y.m(z)), \Gamma_4 \vdash l : \texttt{Removable}}{\Gamma_4 \Rightarrow [m : \texttt{Removable}]\Gamma_4}$$

$$\text{Remove-Class}\ \frac{nonFrameworkClass(k) \quad \forall(x :=^l new\ k(y)), \Gamma_4 \vdash l : \texttt{Removable}}{\Gamma_4 \Rightarrow [k : \texttt{Removable}]\Gamma_4}$$

$$\text{Remove-Stmts}\ \frac{\Gamma_4 \vdash m : \texttt{Removable} \quad l \in statementsIn(m)}{\Gamma_4 \Rightarrow [l : \texttt{Removable}]\Gamma_4}$$

$$\text{Remove-Methods}\ \frac{\Gamma_4 \vdash k : \texttt{Removable} \quad m \in methodsIn(k)}{\Gamma_4 \Rightarrow [m : \texttt{Removable}]\Gamma_4}$$

Fig. 11. Rules for removing code elements.

TABLE II
EXPERIMENT RESULTS.

App	Data Usage			CPU Total			Wi-Fi Running			Computed Power Use		
	Original	Modified	Reduction	Original	Modified	Reduction	Original	Modified	Reduction	Original	Modified	Reduction
WeatherBug	16.84MB	15.33MB	9.0%	2m05s	1m59s	4.8%	8m02s	5m48s	27.8%	5mAh	4mAh	20.0%
Dictionary.com	6.21MB	3.20MB	48.5%	4m22s	3m56s	9.9%	32s	21s	34.4%	8mAh	7mAh	12.5%
Baidu iKnow	3.86MB	2.42MB	37.3%	2m08s	1m18s	39.1%	1m07s	1m06s	1.5%	5mAh	3mAh	40.0%
Walmart	29.67MB	23.02MB	22.4%	8m08s	7m58s	2.0%	1m50s	1m03s	42.7%	17mAh	14mAh	17.6%
Macy's	30.09MB	24.32MB	19.2%	6m06s	4m35s	24.9%	22s	21s	4.5%	11mAh	8mAh	27.3%
China Calendar	8.49MB	7.26MB	14.5%	9m35s	6m24s	33.2%	32s	24s	25.0%	16mAh	11mAh	31.3%
Fox News	8.71MB	5.52MB	36.6%	6m56s	6m43s	3.1%	27s	19s	29.6%	21mAh	18mAh	14.3%
CBS News	9.55MB	5.21MB	45.4%	8m00s	2m40s	66.7%	38s	27s	28.9%	17mAh	5mAh	70.6%
AP Mobile	9.63MB	7.09MB	26.4%	4m55s	1m44s	67.4%	21s	12s	42.9%	12mAh	6mAh	50.0%
Tattoo My Photo	12.79MB	10.04MB	21.5%	10m13s	1m00s	90.2%	41s	31s	24.2%	25mAh	2mAh	92.0%
Average			28.1%			34.1%			26.2%			37.6%

removable UI elements that are correlated to network access and measure the incurred data usage before and after their removal. Similarly, to evaluate the battery usage, we measure the app's total CPU time, Wi-Fi running time and computed power use before and after the code reduction. We obtain this detailed information through the Setting app. We present the experiment results in Table II.

From Table II, we observe that removing some unwanted functionalities and associated UI elements has a minimum reduction of 9.0% for data usage (WeatherBug), 2.0% for total CPU time (Walmart), 1.5% for Wi-Fi running time (Baidu iKnow) and 12.5% for the computed power use (Dictionary.com). The maximum reductions for the four factors are 48.5%, 90.2%, 42.9% and 92.0%, respectively. Eventually, we obtain average reductions of 28.1%, 34.1%, 26.2% and 37.6% for the four factors, which demonstrate the effectiveness of our approach.

Figure 12 depicts the time required to analyze the 10 apps alongside their DEX code size. As shown, the analysis time increases as the code size increases. We have also observed that apps with more complex removable functionalities take longer time to be analyzed. For example, the WeatherBug app (see Section II) requires more than 10 minutes to finish the analysis while the China Calendar app, in which a button is disabled, only needs one and a half minutes.

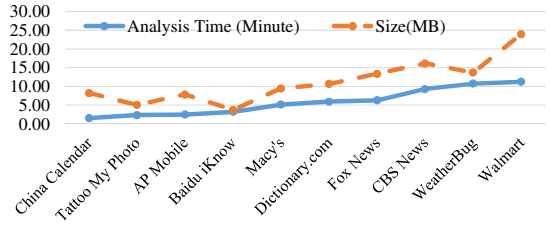

Fig. 12. App analysis time and the DEX code size.

(a) App Dictionary.com (b) App China Calendar

Fig. 13. Screenshots of Dictionary.com and China Calendar.

B. Case Studies

In the following, we use a few cases to illustrate the application of our approach.

1) Removing Unwanted App Components: We use *unwanted app components* to refer to those UI elements that display information provided by the app providers (not third-party advertisement) but are probably unwanted by the users, *e.g.,* the WEATHER NEWS in the WeatherBug app (Figure 1). Here, we study one more case from the Dictionary.com app in this section.

As shown in Figure 13a, the main page of the app contains a search box, a tag for the word of the day and a list of other topics about language, grammar, *etc*. Usually, a user launches the app to look for words and does not expect to be distracted by the other irrelevant contents, which inevitably consume additional data and battery. In our experiment, we remove the irrelevant contents (blog and slideshow features) from the app, which are populated through some background services. Specifically, TOFU first locates the program locations firing the background actions and then disables their invocations. As a result, the blog and slideshow tags are prevented from being displayed on the main page, leading to 48.5% data usage reduction and 12.5% power usage reduction.

2) Removing Unwanted User Actions: The China Calendar app (*com.veryapps.hl*) helps users explore the lunar calendar (*e.g.,* current lunar day, auspicious information, *etc.*). However, the app contains a button in the main page, which once clicked, launches a new page acting similar to a news app (Figure 13b). Accidentally clicking the button will cause unexpected data usage and battery consumption. The implementation is as follows:

```
1 // in class MainActivity
2 public void onCreate(Bundle b) {//omitted operations
3   btn = findViewById(R.id.btn_news_category);
4   btn.setOnClickListener(this);
5 }
6 public void onClick(View v) {
7   Intent i = new Intent(this, NewsCategoryActivity.class);
8   startActivity(i);
9 }
```

The button handler is simplified in method `onClick` at line 6, in which a new activity presenting the news is started (line 8). The unwanted UI element is referred to at line 3. TOFU forwardly tracks the uses of the UI element and types lines 3 and 4. It further tracks and types the button handler and the launched activity. After reduction (*e.g.,* lines 3 and 4), the functionality of the button is disabled. Based on our experiment, the removal saves 14.5% of data usage, 33.2% of CPU time, 25% of Wi-Fi running time and 31.3% of power use when we click the button in original app once per minute for 10 minutes.

3) Removing Advertisements: Advertisements are commonly used in Android apps for monetization purposes. However, mobile advertisements have long been considered a source of distraction for users [11], responsible for additional network data and power consumption, not to mention the risks of privacy leaks and potentially malicious behaviors exhibited by advertisement libraries [4], [5], [6].

Banner ad, displayed usually at the bottom of the screen (see Figure 1), is the most common type of mobile advertisement. As shown in Table II, removing the banner ads in apps (CBS

```
1  class ArticleFragment ... {
2    LoaderCallbacks callback; OAdapter madapter;
3    ArrayList contents;
4    ArticleFragment() {
5      callback = new LoaderA(this);
6      madapter = new OAdapter(...);
7    }
8    void onActivityCreated(Bundle b) {
9      loaderManager.initLoader(..., callback);
10   }
11 }
12 class LoaderA implements LoaderCallbacks {
13   ArticleFragment fragment;
14   Loader onCreateLoader(int i, Bundle b) {
15     return new OLoader(...);
16   }
17   void onLoadFinished(Loader l, Object d) {
18     fragment.contents = (ArrayList) d;
19     fragment.madapter.updateContent((ArrayList)d);
20   }
21 }
22 class OAdapter ... {
23   ArrayList alist;
24   void updateContent(ArrayList list) {
25     alist.addAll(list);
26     this.notifyDataSetChanged();
27   }
28   View getView(int pos, View v, ViewGroup vg) {
29     vv = inflater.inflate(...);//unwanted UI
30     content = alist.get(pos);
31     vv.set(content);//detail omitted
32   }
33 }
```


Fig. 14. Code snippet for Fox News.

News, AP Mobile and Tattoo My Photo) has led to a significant reduction of data usage and battery consumption.

App Fox News adopts another type of advertisement, which displays several pieces of ad titles and images. We noticed that such advertisement is heavily integrated with the app code, compared with the simple banner ad, as shown in Figure 14. We also show in the same figure what the correlated statements are typed with. For example, in their own calling context, line 29 is typed with `UIRelated` in Γ_1 and line 18 is typed with `UIData` in Γ_2 and `InputUIData` in Γ_3. The classes `LoaderA` and `OAdapter` are only used as shown in the code snippet and thus all the typed statements are removable. Our analysis further locates the trigger for the background functionality at line 9 to disable its invocation. According to Table II, removing the unwanted advertisement in this app results in 36.6% of data usage reduction and 14.3% of power use reduction.

C. Discussion

As mentioned in the introduction, our technique is intended to help the development team automatically generate various customizations and personalizations of an app, at the end of development cycle. It is also desirable for retrofitting legacy apps whose source code or original developers are no longer available. For instance, instead of patching a security vulnerability in a legacy app, the maintenance team may choose to remove the corresponding feature. While it is a valid concern that disabling some features such as advertisements may have negative impact on the income of the development team, we anticipate that an alternative business model is for the development team to sell customized versions at a higher price to compensate for the loss. A more aggressive business model

may be to sell the right of customization (using a technique like ours that is certified by authorities) to the end users (*e.g.,* enterprises) such that the customized versions will be signed by the original development team or some authorized party on their behalf to avoid possible legal issues.

As an initial effort of customizing apps by removing features, our technique is limited to reducing features associated with given UI elements. We expect in the future more techniques can be developed to customize/reduce other app features (*e.g.,* background tasks not related to UIs).

In addition, our analysis inherits the limitations of Android app analysis [12], [13], [14]. Specifically, our approach requires a precise and complete call graph which is usually not satisfied in Android app analysis. Context-sensitive analysis, as depicted in Section III, is resource intensive, which inevitably hinders arbitrary app code reduction. Besides, the underlying rewriting framework Soot limits the capability of our prototype. Even though all the 10 apps run well without errors after code reduction, we found that some other apps (*e.g.,* CNN) don't work correctly after the transformation of DEX code by Soot, even if we remove nothing in the code. In addition, removing UI elements (*e.g.,* advertisements) that are embedded in HTML pages rendered by WebView is not supported.

While our technique has usability in consideration to begin with as it allows users to express their needs through the UI, which is far more intuitive than some formal specification language, the implementation is still a research prototype whose user interface has many places to improve.

V. Related Work

Techniques have been proposed for detecting arbitrary third-party libraries, including ad libraries, in Android apps. Some rely on simple techniques such as white-listing namespaces of popular ad libraries [15], [16]. Narayanan *et al.* distinguished primary and non-primary modules of apps through hierarchical clustering [17] whereas Liu *et al.* developed a classifier for ads detection based on code features and package relationship [18]. LibRadar [19] relies on API frequencies. Along the same line, Backes *et al.* proposed an obfuscation resilient ad detection technique, through extracting profiles resilient to common obfuscation techniques and relying on class-hierarchy rather than code [20]. In comparison, our technique is more general. It focuses on removing features indicated by users on the UI, not just ads. It does not rely on specific code patterns.

There are also techniques that detect redundancy in applications [21], [22], [23], [24]. Their purpose is to make the software systems more resilient to failures and to leverage duplicated code as test oracles. They are complementary to our technique as redundant code can be removed/replaced for better quality. They discover redundant code starting from faulty components in programs while our approach starts from user specification of UI information.

In addition, a lot of works aim at improving energy efficiency. Gottschalk *et al.* detected and removed energy-wasting code in Android apps with the knowledge of energy-inefficiency patterns [25], [26]. Sahin *et al.* studied energy consumption caused by code obfuscation in mobile apps [27]. Gui *et al.* measured energy consumption of mobile ads [28]. Banerjee *et al.* proposed a technique of refactoring Android apps to enhance energy efficiency following a set of guidelines [29]. Wu *et al.* statically detected energy defects in app UIs with predefined energy draining patterns [30]. Our proposed approach doesn't rely on code patterns, nor does it require domain knowledge to work properly. Energy saving occurs because of the removal of selected unwanted features. Martins *et al.* presented TAMER to improve battery lifetime by instrumenting the Android OS and interposing events and signals that cause background task wakeups [31]. Linares-Váseuez *et al.* [32] and Li *et al.* [33] proposed approaches for reducing display energy through automatically changing the color schemes.

There are works using UI information to detect attacks or privacy leaks. AsDroid [34] uses UI information to confirm whether a program behavior is expected by the app user. SUPOR [35], UIPicker [36] and BidText [37] utilize UI information to check data sensitiveness displayed or entered on UI. In particular, BidText [37] features a type system for information disclosure detection by recognizing the text information as types and performs type propagation along data flows. The type-based taint analysis system developed by Ernst *et al.* predefines a few security types and checks if types reach a program point are compatible [38].

VI. Conclusion

We propose a static technique to remove code elements in Android apps. The code elements are relevant to user specified unwanted UI elements. The approach identifies the program locations directly referring to the specific UI elements and applies a type system to infer removable code elements. Each reachable code element is tagged with a type that is propagated. The types are used to determine whether the corresponding code elements are removable or not. In addition to removing code elements that are related to the specified UI elements, our technique is also able to discover the associated background functionalities and type the corresponding code elements in the background functionalities such that they can be removed too. We implement a prototype and evaluate it on 10 real-world Android apps. The results show that our approach can accurately identify removable code elements associated with the specified UI elements and removing those functionalities leads to substantial resource savings.

Acknowledgments

We thank the anonymous reviewers for their constructive comments. This research was supported, in part, by DARPA under contract FA8650-15-C-7562, NSF under awards 1409668, 1320444, and 1320306, and ONR under contracts N000141410468 and N000141712947. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] Search Engine Watch, "Mobile now exceeds PC: The biggest shift since the Internet began," https://searchenginewatch.com/sew/opinion/2353616/mobile-now-exceeds-pc-the-biggest-shift-since-the-internet-began.

[2] A. Brain, "Statistics of Android ad networks," https://www.appbrain.com/stats/libraries/ad.

[3] J. Boutet, "Malicious Android applications: Risks and exploitation," https://www.sans.org/reading-room/whitepapers/threats/malicious-android-applications-risks-exploitation-33578.

[4] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "PiOS: Detecting privacy leaks in iOS applications," in *Proceedings of the Network and Distributed System Security Symposium*, ser. NDSS 2011.

[5] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI 2010.

[6] V. Rastogi, R. Shao, Y. Chen, X. Pan, S. Zou, and R. Riley, "Are these ads safe: Detecting hidden attacks through the mobile app-web interfaces," in *Proceedings of the Network and Distributed System Security Symposium*, ser. NDSS 2016.

[7] J. Gui, S. Mcilroy, M. Nagappan, and W. G. J. Halfond, "Truth in advertising: The hidden cost of mobile ads for software developers," in *Proceedings of the 37th International Conference on Software Engineering*, ser. ICSE 2015.

[8] Earth Networks, "Weather by WeatherBug," https://play.google.com/store/apps/details?id=com.aws.android.

[9] T. Book, "Privacy concerns in Android advertising libraries," in *Master Thesis*, 2013.

[10] "Soot: A framework for analyzing and transforming Java and Android applications," http://sable.github.io/soot/.

[11] M. An, "Why people block ads (and what it means for marketers and advertisers)," https://research.hubspot.com/reports/why-people-block-ads-and-what-it-means-for-marketers-and-advertisers.

[12] Y. Wang, H. Zhang, and A. Rountev, "On the unsoundness of static analysis for Android GUIs," in *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, ser. SOAP 2016.

[13] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "Droidra: Taming reflection to support whole-program analysis of Android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016.

[14] L. Li, T. F. Bissyand, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of Android apps," *Inf. Softw. Technol.*, vol. 88, no. C, Aug. 2017.

[15] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WISEC 2012.

[16] T. Book, A. Pridgen, and D. S. Wallach, "Longitudinal analysis of Android ad library permissions," *CoRR*, vol. abs/1303.0857, 2013.

[17] A. Narayanan, L. Chen, and C. K. Chan, "Addetect: Automated detection of Android ad libraries using semantic analysis," in *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing*, ser. ISSNIP 2014.

[18] B. Liu, B. Liu, H. Jin, and R. Govindan, "Efficient privilege de-escalation for ad libraries in mobile apps," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys 2015.

[19] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: Fast and accurate detection of third-party libraries in Android apps," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE 2016.

[20] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in Android and its security applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS 2016.

[21] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè, "Automatic workarounds for web applications," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2010.

[22] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè, "Automatic recovery from runtime failures," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE 2013.

[23] A. Carzaniga, A. Goffi, A. Gorla, A. Mattavelli, and M. Pezzè, "Cross-checking oracles from intrinsic software redundancy," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014.

[24] A. Goffi, A. Gorla, A. Mattavelli, M. Pezzè, and P. Tonella, "Search-based synthesis of equivalent method sequences," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014.

[25] M. Gottschalk, M. Josefiok, J. Jelschen, and A. Winter, "Removing energy code smells with reengineering services," in *Informatik 2012*.

[26] J. Jelschen, M. Gottschalk, M. Josefiok, C. Pitu, and A. Winter, "Towards applying reengineering services to energy-efficient applications," in *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*, ser. CSMR 2012.

[27] C. Sahin, M. Wan, P. Tornquist, R. McKenna, Z. Pearson, W. G. Halfond, and J. Clause, "How does code obfuscation impact energy usage?" *Journal of Software: Evolution and Process*, vol. 28, no. 7, pp. 565–588, 2016.

[28] J. Gui, D. Li, M. Wan, and W. G. Halfond, "Lightweight measurement and estimation of mobile ad energy consumption," in *Proceedings of the International Workshop on Green and Sustainable Software*, ser. GREENS 2016.

[29] A. Banerjee and A. Roychoudhury, "Automated re-factoring of Android apps to enhance energy-efficiency," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft 2016.

[30] H. Wu, S. Yang, and A. Rountev, "Static detection of energy defect patterns in Android applications," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016.

[31] M. Martins, J. Cappos, and R. Fonseca, "Selectively taming background Android apps to improve battery lifetime," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC 2015.

[32] M. Linares-Vásquez, G. Bavota, C. E. B. Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Optimizing energy consumption of guis in Android apps: A multi-objective approach," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015.

[33] D. Li, A. H. Tran, and W. G. J. Halfond, "Optimizing display energy consumption for hybrid Android apps (invited talk)," in *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, ser. DeMobile 2015.

[34] 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*, ser. ICSE 2014.

[35] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang, "Supor: Precise and scalable sensitive user input detection for Android apps," in *Proceedings of the 24th USENIX Security Symposium*, ser. USENIX Security 2015. [Online]. Available: http://blogs.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/huang

[36] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang, "Uipicker: User-input privacy identification in mobile applications," in *Proceedings of the 24th USENIX Security Symposium*, ser. USENIX Security 2015.

[37] J. Huang, X. Zhang, and L. Tan, "Detecting sensitive data disclosure via bi-directional text correlation analysis," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016.

[38] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. B. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu, "Collaborative verification of information flow for a high-assurance app store," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS 2014.